VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV MIKROELEKTRONIKY
DEPARTMENT OF MICROELECTRONICS

BEHAVIORÁLNÍ SYNTÉZA DIGITÁLNÍCH OBVODŮ
HIGH-LEVEL SYNTHESIS OF DIGITAL CIRCUITS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Ján Jendrušák**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Vojtěch Dvořák**
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Mikroelektronika a technologie**
Ústav mikroelektroniky

Student: Ján Jendrušák ***ID:*** 174320
Ročník: 3 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Behaviorální syntéza digitálních obvodů

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s nástrojem Cadence Stratus HLS pro behaviorální syntézu (HLS). Prostudujte známé architektury FFT pro implementaci do obvodů FPGA a ASIC. Algoritmus FFT popište pomocí knihovny SystemC a proveďte syntézu pomocí nástroje pro HLS. V práci popište metodiku správného návrhu digitálních obvodů s využitím HLS.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce

Termín zadání: 6.2.2017 ***Termín odevzdání:*** 8.6.2017

Vedoucí práce: Ing. Vojtěch Dvořák
Konzultant:

doc. Ing. Jiří Háze, Ph.D.
předseda oborové rady

ABSTRAKT

Táto práca sa zaoberá praktickým otestovaním behaviorálnej syntézy ako spôsobu návrhu digitálnych obvodov a jej momentálnym progresom pri tvorbe RTL popisov. V úvode práce sú popísané hlavné úlohy behaviorálnej syntézy spolu s knižnicou tried jazyka C++ nazvanou SystemC, ktorá implementuje hardvérové konštrukcie, dátové typy s definovateľnou dátovou šírkou a vie pracovať s časom. Ďalej sa práca zameriava na diskrétnu Fourierovu transformáciu a jej modifikáciu pre efektívnejší výpočet – rýchlu Fourierovu transformáciu. V praktickej časti práce je navrhnutý referenčný model algoritmu FFT, ktorý je ďalej vhodne upravený a prevedený nástrojom pre behaviorálnu syntézu Stratus High-Level Synthesis do viacerých hardvérových architektúr.

KĽÚČOVÉ SLOVÁ

ASIC, behaviorálna syntéza, diskrétna Fourierova transformácia, FPGA, HLS, RTL, rýchla Fourierova transformácia, Stratus High-Level Synthesis, SystemC

ABSTRACT

This thesis deals with practical test of high-level synthesis as a digital circuits design method and its current progress in creating RTL models. At first main tasks of HLS will be described together with C++ library of classes called SystemC, which implements hardware constructs, notion of time and hardware datatypes with arbitrary bit width. After that thesis focuses on discrete Fourier transform and its fast form of computation – fast Fourier transform. In the practical part of thesis reference FFT model is written in C++ language, which is later edited appropriately a synthesized with Stratus High-Level Synthesis tool into several hardware architectures.

KEYWORDS

ASIC, discrete Fourier transform, fast Fourier transform, FPGA, high-level synthesis, HLS, RTL, Stratus High-Level Synthesis, SystemC

JENDRUŠÁK, J. *Behaviorální syntéza digitálních obvodů*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií. Ústav mikroelektroniky, 2017. 37 s., 16 s. příloh. Bakalářská práce. Vedoucí práce: Ing. Vojtěch Dvořák

PREHLÁSENIE

Prehlasujem, že svoju bakalársku prácu na tému Behaviorální syntéza digitálních obvodů som vypracoval samostatne pod vedením vedúceho bakalárskej práce a s použitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej bakalárskej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto bakalárskej práce som neporušil autorské práva tretích osôb, obzvlášť som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a majetkových a som si plne vedomý následkov porušení ustanovení § 11 a nasledujúcich zákona č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovení časti druhej, hlavy VI. diel 4 Trestného zákonníku č. 40/2009 Sb.

V Brne dňa ....

(podpis autora)

POĎAKOVANIE

Ďakujem vedúcemu bakalárskej práce Ing. Vojtěchu Dvořákovi, za účinnú metodickú, pedagogickú a odbornú pomoc a ďalšie cenné rady pri spracovaní mojej bakalárskej práce.

V Brne dňa ....

(podpis autora)

OBSAH

ÚVOD

Integrované obvody sa od svojich počiatkov na prelome 60. a 70. rokov 20. storočia riadili odhadom Gordona E. Moora – spoluzakladateľa spoločnosti Intel Corporation. Podľa tohto prehlásenia, dnes známeho ako Moorov zákon, sa zložitosť integrovaných obvodov každých 18 mesiacov zdvojnásobí pri zachovaní tej istej ceny [1].

S postupným nárastom komplexnosti sa zároveň zvyšovali aj nároky na úroveň abstrakcie návrhu pre jeho väčšiu efektivitu. Základné stavebné prvky IO – tranzistory, boli preto najprv pri návrhu nahradené logickými hradlami, a neskôr s príchodom logickej syntézy modelmi na úrovni RTL [2].

Behaviorálna syntéza (v nasledujúcom texte bude používaná skratka HLS) je alternatívny spôsob dizajnu digitálnych obvodov, ktorý má potenciál časom nahradiť aktuálne prevládajúcu RTL metodiku návrhu. HLS svojím algoritmickým zápisom požadovanej funkcie so sebou prináša prísľub zjednodušenia návrhu čoraz zložitejších systémov a tým aj ich rýchlejšie uvedenie na trh.

Cieľom tejto bakalárskej práce je prakticky odskúšať momentálne možnosti behaviorálnej syntézy na návrhu algoritmu rýchlej Fourierovej transformácie. Aby bolo možné túto úlohu vykonať, je potrebné sa najskôr zoznámiť so samotným princípom behaviorálnej syntézy, používaným programovacím vybavením, teoretickým rozborom diskrétnej a rýchlej Fourierovej transformácie a samotným nástrojom určeným na behaviorálnu syntézu. V prvej kapitole práce je vysvetlený rozdiel medzi klasickým postupom návrhu a HLS metodikou, tiež sú bližšie popísané jednotlivé kroky, ktoré musí nástroj pre HLS vykonať. Záver kapitoly stručne komentuje vývoj HLS v čase.

Druhá kapitola podrobnejšie rozoberá nadstavbu jazyka C++ - SystemC. Táto nadstavba obsahuje konštrukcie nutné pre modelovanie digitálnych obvodov s ľubovoľnou bitovou presnosťou a je využívaná balíkom Stratus High-Level Synthesis aj inými nástrojmi na behaviorálnu syntézu. Budú predstavené základné prvky SystemC spolu s novými dátovými typmi, ktoré podporujú definovanie bitovej šírky.

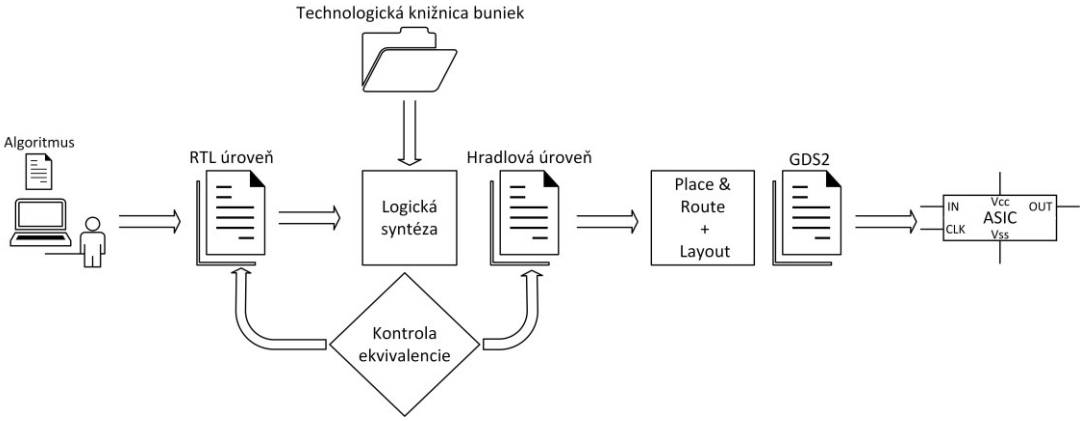
Pred samotnou praktickým návrhom algoritmu rýchlej Fourierovej transformácie bude v ďalšej časti popísaná diskrétna Fourierova transformácia, algoritmus rýchlej Fourierovej transformácie (označovaný aj FFT – podľa anglického názvu fast Fourier transform) slúžiaci na jej efektívny výpočet, základné pojmy dôležité pre jeho pochopenie a jeho grafická reprezentácia. Zároveň budú predstavené aj odlišné architektúry rýchlej Fourierovej transformácie pre obvody FPGA a ASIC. Tretiu kapitolu ukončí navrhnutý referenčný algoritmus FFT v jazyku C++, ktorý bude slúžiť ako predloha pre HLS s nástrojom Stratus High-Level Synthesis.

Posledné dve kapitoly sa venujú samotnému nástroju Stratus High-Level Synthesis od spoločnosti Cadence, ktorý bude využitý na behaviorálnu syntézu algoritmu FFT do RTL podoby. Najskôr je popísaná samotná štruktúra projektu v rámci nástroja a jeho možnosti pri nastavovaní parametrov výstupného RTL kódu. Následne bude referenčný algoritmus modifikovaný s minimálnym počtom úprav tak, aby bol syntetizovateľný. Pre tento kód bude nájdený vhodný súbor nastavení pre získanie rozličných architektúr výpočtu FFT s dôrazom na pochopenie fungovania HLS. Tieto architektúry budú potom medzi sebou porovnané a v závere sa zhodnotí aktuálny stav behaviorálnej syntézy ako metodiky návrhu digitálnych integrovaných obvodov.

1 BEHAVIORÁLNA SYNTÉZA

Behaviorálnu syntézu možno definovať ako automatizovaný proces generovania RTL popisu digitálneho obvodu podľa algoritmického vyjadrenia požadovaného chovania [2]. Práve spomínaná automatizácia tvorby RTL popisu by mala byť vylepšením momentálne typického postupu pri návrhu, ktorý je ukázaný na obr. 1.1.

Na začiatku každého projektu sa väčšinou spíše špecifikácia a požiadavky kladené na návrh, poprípade spustiteľný referenčný model v jazyku ANSI C, C++ alebo SystemC. Tieto modely slúžia len na overenie funkčnosti, určenie žiadaných vlastností a ešte v sebe neobsahujú veľa informácií o hardvérovej podobe systému. Až po otestovaní tohto modelu sa prechádza na výber architektúry, teda spôsobu, ako sa bude daná funkcia vykonávať. Následne je vybraná architektúra navrhnutá tímom dizajnérov v jazyku VHDL alebo Verilog [2].



Obrázok 1.1 RTL metodika návrhu

Vzhľadom na neustály technologický progres a zvyšujúcu sa komplexnosť návrhov však tento krok môže predstavovať dlhotrvajúci cyklus hľadania chýb. High-level syntéza so sebou prináša prísľub urýchlenia celého procesu a zvýšenia produktivity, keďže vstupom pre ňu je kód na vyššej úrovni abstrakcie spolu s možnými nastaveniami na zlepšenie kvality výsledkov. Nástroj EDA spolu s pripojenou technologickou knižnicou súčastí vytvorí z behaviorálneho popisu RTL kód [2][3].

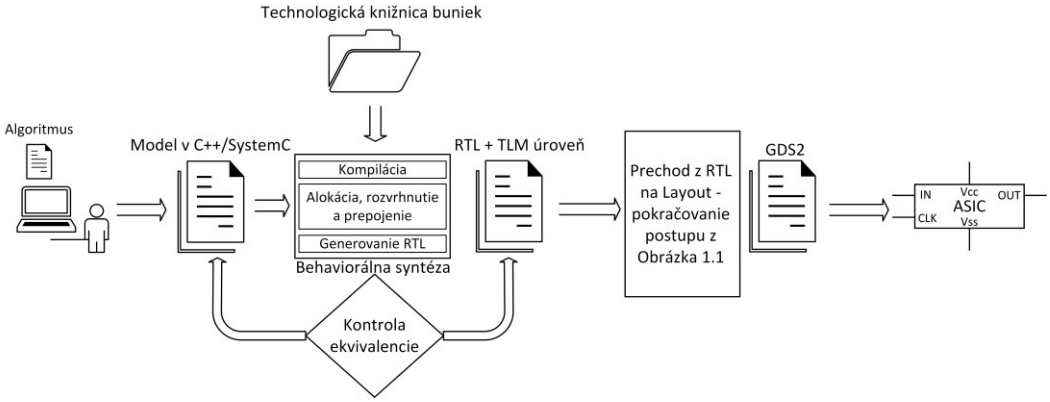
1.1 Úlohy HLS nástroja v novom postupe návrhu

Proces generovania RTL nástrojom pre HLS pozostáva z niekoľkých základných krokov. Na začiatku prebehne kompilácia funkčnej špecifikácie, zároveň zvyčajne prebehnú procesy na optimalizáciu kódu, ako napríklad transformácia slučiek alebo eliminácia nadbytočného kódu [4].

Na základe využitých operácií sú z technologickej knižnice komponentov vybrané hardvérové prostriedky tak, aby splnili návrhové požiadavky. Pre dosiahnutie požadovaného časovania sú potom operácie rozvrhnuté do jednotlivých cyklov. Podľa použitého komponentu z knižnice môže operácia trvať jeden alebo viac cyklov, poprípade sa môžu niektoré operácie vykonávať paralelne, pokiaľ ich dáta nie sú

navzájom závislé. Poslednou úlohou pred samotným generovaním RTL kódu je priradenie premenných k príslušným pamäťovým bunkám alebo registrom a operácií ku kompatibilným funkčným jednotkám. Optimalizácia zdieľania prostriedkov pritom môže byť ponechaná na nástroj HLS, alebo ju môže ovplyvniť používateľ [4].

Ďalšou výhodou HLS metodiky je okamžité overenie funkčnosti algoritmu a aj RTL podoby simuláciou napísanou v jazyku C++ [2]. Nástroje HLS môžu byť priamo prepojené aj s ďalšími nástrojmi (napríklad na logickú syntézu), môžu im byť predané ďalšie nastavenia, a tak môže postup návrhu zobrazený na obr. 1.2 pokračovať podobne ako na obr. 1.1.



Obrázok 1.2 HLS metodika návrhu

1.2 Vývoj behaviorálnej syntézy

Prvé realizácie HLS sa objavovali už v 80. rokoch 20. storočia a boli založené na modifikácii jazyka C. Tieto odlišné varianty úprav však spôsobili roztrieštenosť metódy, čo bol jedným z dôvodov, prečo sa HLS neujala a mala problém sa ďalej rozvíjať. Alternatívou k tomuto prístupu sa stalo rozšírenie objektovo orientovaného jazyka C++ o triedy s potrebnými konštrukciami a novými dátovými typmi [3].

Hlavným zástupcom takejto skupiny je SystemC, ktorý bol predstavený už v 90. rokoch 20. storočia, spočiatku ale nebol dostatočne podporovaný nástrojmi na HLS. Viacej sa začal využívať novou generáciou nástrojov, ktoré prišli na trh na začiatku 21. storočia, a jedným z nich je aj nástroj Stratus High-Level Synthesis [2][3].

2 SYSTEMC

Názov SystemC pomenúva knižnicu tried jazyka C++, ktorá umožňuje vytvárať modely či už softvérových algoritmov, alebo hardvérových implementácií s presným popisom fungovania v každom hodinovom cykle. S využitím SystemC je možné vytvoriť spustiteľnú špecifikáciu požadovaného systému, ktorá sa neskôr môže použiť na samotnú hardvérovú implementáciu [5].

Spojenie programovacieho jazyka C++, ktorý poskytuje dostatočnú úroveň abstrakcie, a konštrukcií zabezpečujúcich časovanie, súbežnosť a citlivosť na určité zmeny v podobe knižnice SystemC, vytvorilo vhodné riešenie pre momentálne návrhárske potreby. C++ je tiež veľmi rozšíreným jazykom, a keďže bolo minimalizované použitie jeho špecifických konštrukcií, pri učení sa SystemC veľmi pomôže aj znalosť jazyka C [5].

Aktuálne je SystemC podporovaný nezávislou organizáciou Accelera Systems Initiative a od svojho prvého vydania prešiel rozsiahlymi úpravami. Výraznou zmenou bol prechod na verziu 2.0 (a neskôr ďalšie aktualizácie), ktorá dáva priestor efektívnejšiemu modelovaniu na úrovni transakcií [5][6].

2.1 Hierarchia navrhovaného systému

Základným blokom v návrhu napísanom pomocou SystemC sú moduly. Tie slúžia na rozdelenie dizajnu na samostatné funkčné bloky, ktoré môžu byť medzi sebou poprepájané. Každý modul v sebe môže obsahovať prvky implementujúce požadovanú funkciu [5]. Pozitívom tejto koncepcie systému je podobnosť s HDL jazykmi používanými na popis RTL modelov, ako je napríklad Verilog alebo VHDL.

2.1.1 Procesy

Procesy zabezpečujú žiadanú funkciu modulu. Možno ich prirovnať ku metódam jazyka C++, ktoré sú rozpoznateľné jadrom SystemC. Proces sa spustí pri zmene signálu, ktorý je súčasťou jeho citlivostného zoznamu a kľúčové slovo pre definovanie modulu je SC_MODULE. Po spustení procesu je kód v jeho vnútri vykonávaný sekvenčne [5].

Jednotlivé procesy modulu sa definujú v konštruktore, ktorý má za úlohu vytvoriť a inicializovať inštanciu modulu. Konštruktor je tak akýmsi základným stavebným prvkom modulu a deklaruje sa vnútri modulu kľúčovým slovom SC_CTOR [5].

Existujú tri typy procesov, ktoré SystemC podporuje, a prvým z nich je Method Process. Keď je tento proces spustený udalosťou na definovanom signáli, vykoná svoju funkciu a vráti ovládanie naspäť simulačnému jadru. Method Process pritom nemôže volať funkciu, ktorá v sebe obsahuje *wait()* príkaz, teda nemôže byť znovu spustený. Kľúčové slovo pre tento proces je SC_METHOD [5]. Method Process je používaný hlavne na tvorbu kombinačnej logiky, ako môžeme vidieť aj na jednoduchom príklade modulu dvojvstupového logického hradla typu AND (použité porty, ich metódy a dátové typy sú popísané v kapitolách 2.1.2 a 2.2):

```
#include <systemc.h>
SC_MODULE(and2){
sc_in<sc_bool>    a, b;             // ports declaration
sc_out<sc_bool>   c;
void and_gate(){                    // function of module
      c.write( a.read() & b.read() );
}
SC_CTOR(and2){
      SC_METHOD(and_gate);
      sensitive << a << b;                // sensitivity list
}
}
```

Ďalším predstaviteľom je Thread Process, ktorého hlavnou odlišnosťou od Method Processu je jeho reaktivácia. Môže tým pádom obsahovať *wait()* funkciu, ktorá pozastaví vykonanie procesu, kým sa znovu objaví definovaná udalosť, ktorá ho spustí od toho bodu, v ktorom sa naposledy zastavila. Telo funkcie tohto procesu, ktorý sa v konštruktore deklaruje kľúčovým slovom SC_THREAD, sa vkladá do nekonečnej slučky [5]. Pomocou SC_THREAD môže byť popísaný napríklad jednoduchý osembitový čítač s povoľovacím signálom *enable*:

```
#include <systemc.h>
SC_MODULE(counter){
sc_in_clk         clk;
sc_in<sc_bool>    rst;
sc_in<sc_bool>    enable;
sc_out<sc_uint<8> >     count_out;
sc_uint<8>        count_int;  // local variable
void up_count(){
  while(true){
            wait();                 // waiting for event - clk.pos()
            if(rst.read() == 1){
                  count_int = 0;
            } else if (enable.read() == 1){
                  count_int = count_int + 1;
            }
            count_out.write(count_int);
}
SC_CTOR(counter){
      SC_THREAD(up_count, clk.pos());
      sensitive << clk.pos();
}
}
```

Špeciálnym prípadom Thread Processu je Clocked Thread Process s definovaným slovom SC_CTHREAD. Ako napovedá samotný názov, tento proces je spustený len v prípade definovanej hrany hodinového signálu, čo odpovedá spôsobu implementácie hardvéru. SC_CTREAD teda nemá samostatný citlivostný zoznam. [5] Pokiaľ budeme chcieť popísať ten istý čítač ako v poslednom príklade, bude mať SC_CTHREAD odlišný konštruktor:

```
SC_CTOR(counter){
SC_CTHREAD(up_count, clk.pos());
}
```

2.1.2 Prepojenie modulov

Aby mohol modul komunikovať s ďalšími modulmi, sú mu priradené modulové porty. Každý port môže byť vstupný – *sc_in*, výstupný – *sc_out*, alebo vstupno-výstupný – *sc_inout*. Na prístup k portom sa využívajú základné metódy *write()* a *read()*. Medzi porty sú potom pripojené signály označené kľúčovým slovom *sc_signal*, ktoré prenášajú dáta, a porty určujú ich smer. Signály smú byť aj lokálneho charakteru, teda sa môžu nachádzať vnútri modulu [5].

Na ukážku mapovania signálov a portov využijeme zjednodušený príklad modulu *top,* ktorý sa môže použiť na behaviorálnu simuláciu a obsahuje modul *tb* a testovaný modul *dut.* Pred samotným spojením signálov a portov sa v module deklarujú potrebné signály a ukazovatele pre každý objekt, ktorého inštancia bude neskôr vytvorená. V našom prípade časť kódu v module bude vyzerať takto:

```
sc_signal <sc_uint<16> >      inp_sig, outp_sig;
sc_clock                      clk;
sc_signal<bool>               rst;
tb    *tb1;
dut   *dut1;
```

Následne sú v konštruktore vytvorené inštancie objektov a prepojenie medzi nimi sa môže vytvoriť buď skráteným zápisom, pri ktorom sa musí dbať na poradie priradzovaných signálov podľa deklarácie portov, alebo je každý port prepojený s korešpondujúcim signálom samostatne [5][8]. Druhý spôsob môžeme vidieť v nasledujúcom zdrojovom kóde, ktorý je súčasťou konštruktoru modulu *top*:

```
// connecting the device under test module
dut1 = new dut("dut1");
dut1->clk(clk);
dut1->rst(rst);
dut1->inp(inp_sog);
dut1->outp(outp_sig);
// connecting the testbench
tb1 = new tb("tb1");
tb1->clk(clk);
tb1->rst(rst);
tb1->outputs(inp_sig);
tb1->inp (outp_sig);
```

2.1.3 Vnútorná úschova dát

Uschovať dáta v rámci modulu je možné deklaráciami vnútorných premenných, ako bolo ukázané v príkladoch podkapitoly Procesy. Podporované sú dátové typy jazyka C++ aj SystemC, a tieto dáta sú viditeľné iba pre modul, v ktorom sa nachádzajú [5]. Pri hardvérovej implementácii potom závisí na nástroji na HLS, ako tieto dáta interpretuje, alebo túto voľbu ponechá na návrhárovi.

2.2 Dátové typy

Ako už bolo naznačené v kapitole 2.1.2, SystemC podporuje natívne dátové typy C++, no zároveň pridáva vlastné dátové typy, ktoré vernejšie reprezentujú následnú hardvérovú implementáciu. Pri natívnych dátových typoch C++ je nutné

si dávať pozor na isté obmedzenia najmä čo sa týka možností následnej high-level syntézy. Napríklad dátové typy s pohyblivou desatinnou čiarkou sú podporované pre behaviorálnu simuláciu a pre inicializáciu syntetizovateľných dátových typov, pre samotnú behaviorálnu syntézu to však neplatí. Prekážkou použitia týchto dátových typov je odlišnosť zachádzania s nimi u jednotlivých kompilátorov [8].

2.2.1 Celočíselné dátové typy s definovateľnou bitovou šírkou

SystemC umožňuje deklarovať celočíselné dátové typy so znamienkom aj bez znamienka, podľa čoho je potom určený aj číselný rozsah určitej premennej [7].

Celočíselné dátové typy so znamienkom majú označenie *sc_int* a sú reprezentované tzv. dvojkovým doplnkom. Dátový typ bez znamienka nesie označenie *sc_uint* a oba majú spoločný limit šírky 64 bitov. V prípade potreby väčších celých čísel sú k dispozícii dátové typy *sc_bigint* a variant bez znamienka *sc_biguint* [7].

Všetky spomenuté dátové typy podporujú základné operácie a konverzie medzi sebou navzájom. Tiež smú byť použité v kombinácii s celočíselnými alternatívami jazyka C++. V prípade, že má výsledok operácie väčší počet bitov ako je premenná, do ktorej je priradzovaný, sú ostatné bity mimo rozsah odstránené [5].

2.2.2 Dátové typy s pevnou desatinnou čiarkou

Minimalizácia potrebného hardvéru sa dá dosiahnuť náhradou čísel s pohyblivou desatinnou čiarkou za pevnú. Preto SystemC obsahuje štyri základné typy určené na modelovanie týchto čísel – *sc_fixed*, *sc_ufixed, sc_fix* a *sc_ufix*. Písmeno „u“ v názve znova označuje variantu bez znamienka [5].

Rozdiel medzi prvou a druhou spomenutou dvojicou je v ich argumentoch. Argumenty typov *sc_fixed* a *sc_ufixed* musia byť známe v čase kompilácie, zatiaľ čo *sc_fix* a *sc_ufix* môžu využívať premenné na určenie dĺžky slova. Z tohto rozdielu vyplýva, že pre syntézu sú potenciálne vhodné prvé dva spomínané dátové typy, teda *sc_fixed* a *sc_ufixed* [5][8].

Pre všetky štyri dátové typy je možné nastaviť dĺžku slova, dĺžku celočíselnej časti a chovanie pri priradzovaní čísla s väčšou bitovou šírkou v celočíselnej oblasti. Vtedy sa jedná o tzv. mód pretečenia. Pokiaľ sa priradzuje číslo s väčšou bitovou šírkou v desatinnej časti, hovorí sa o tzv. kvantizačnom móde [5].

Pri nešpecifikovaní niektorého z parametrov sú použité predvolené hodnoty danej triedy. Pre kvantizačný mód je zvolené predvolené nastavenie SC_TRN, kedy sa v prípade potreby výsledok zaokrúhli vždy smerom dole k najbližšej možnej hodnote. Pre prípad pretečenia je okrem samotného módu možné nastaviť aj počet saturovaných bitov. Predvolenými nastaveniami riešenia pretečenia je mód SC_WRAP a počet saturovaných bitov je rovný nule. Pri týchto nastaveniach sa pri pretečení odstránia bity, ktoré sa nachádzajú mimo nastaveného rozsahu celočíselnej časti.

2.2.3 Logické dátové typy a vektory

Zástupcami logických dátových typov sú *sc_bit* a *sc_logic*. Typ *sc_bit* môže nadobúdať buď hodnotu ‘0’ alebo ‘1’ a je dovolené ho používať spolu s logickými typmi jazyka C++.

K typu *sc_logic* môže byť okrem týchto hodnôt priradená aj hodnota ‘X’ (neznáma hodnota), alebo hodnota ‘Z’, teda vysoká impedancia. Pre syntézu však tieto dve hodnoty nie sú vhodné. Dátové typy *sc_bv* a *sc_lv* sú v podstate polia alebo vektory dátových typov *sc_bit*, respektíve *sc_logic* [5][8].

3 DISKRÉTNA FOURIEROVA TRANSFORMÁCIA A ALGORITMUS FFT

Diskrétna Fourierova transformácia (ďalej v texte označovaná skratkou DFT) je formou číselného spracovania signálov, inými slovami slúži na analýzu postupnosti vzorkovaných hodnôt určitého procesu meniaceho sa v čase. Vychádza pritom zo spojitej formy Fourierovej transformácie a prevádza túto časovú postupnosť na postupnosť vo frekvenčnej oblasti. Jej predpis v exponenciálnom tvare je

$$X[m]=\sum_{n=0}^{N-1} x[n]*e^{-j*2\pi*n*m/N}, \tag{3.1}$$

kde *X[m]* je *m*-tá hodnota postupnosti z frekvenčnej domény (celá postupnosť je označovaná aj ako spektrum), *x[n]* je *n*-tou vzorkou časovej postupnosti, *m* označuje poradie výstupnej vzorky, *n* poradie vstupnej vzorky a *N* je celkový počet vstupných vzoriek. Veľkosť vstupného vektoru *N* zároveň predurčuje aj veľkosť výslednej postupnosti vo frekvenčnej oblasti [9].

Dôvodom používania DFT sú ďalšie možnosti spracovania signálov a zjednodušenie niektorých operácií pri práci so spektrom signálu. Napríklad operácia kruhovej diskrétnej konvolúcie dvoch signálov v časovej oblasti, ktorej prevedenie je výpočtovo veľmi náročné, sa vo frekvenčnej doméne mení na súčin dvoch spektier [10]. Akákoľvek analýza číslicových signálov s pomocou DFT by nebola možná bez jej jednej kľúčovej vlastnosti - linearity. Pre DFT teda platí, že spektrum získané zo súčtu dvoch signálov v časovej oblasti je rovné spektru získanému zo súčtu dvoch samostatných DFT signálov [9][11].

Problémom DFT je však vysoký počet jednotlivých operácií. Už pri pohľade na rovnicu 3.1 môžeme vidieť, že s narastajúcim počtom vstupných vzoriek N sa zvyšuje počet operácií násobenia aj sčítania, konkrétne pre násobenie je to kvadratická závislosť N^2 a počet operácií sčítania je rovný $N*(N-1)$. Práve kvôli náročnosti výpočtov nebola najprv DFT použiteľná v praxi, až kým v roku 1965 James Cooley a John Tukey nepublikovali článok o efektívnom spôsobe implementácie diskrétnej Fourierovej transformácie a nazvali ho rýchla Fourierova transformácia. Od ich prevratného objavu bolo popísaných množstvo alternatívnych spôsobov, ako zredukovať počet operácií DFT. Nie všetky algoritmy sú ale vhodné pre určitú implementáciu a algoritmus Cooley – Tukey je tak dodnes veľmi populárnym a využívaným variantom výpočtu diskrétnej Fourierovej transformácie [9].

3.1 Rýchla Fourierova transformácia

Ako bolo spomenuté v úvode kapitoly 3, rýchla Fourierova transformácia je dnes súhrnným označením skupiny algoritmov počítajúcich DFT efektívnejšie ako základný vzťah 3.1. Táto práca sa bude konkrétne zaoberať algoritmom Cooley – Tukey. Vo všeobecnosti je tento algoritmus najuniverzálnejším algoritmom FFT, pretože každé číslo *N* je možné rozdeliť na súčin dvoch celých čísel, podľa ktorých sa výpočet rozdelí na menšie DFT, ktoré sa potom vynásobia vhodnými konštantami. Najviac používané

sú však varianty s počtom vzoriek *N* rovným mocnine čísla 4 a 2 označované radix-4, respektíve radix-2 [12].

Pozrime sa bližšie, akým spôsobom typ FFT algoritmu radix-2 umožňuje znížiť počet operácií. Ako bolo naznačené, je najprv vstupný vektor rozdelený na polovice na párne a nepárne vzorky. Pomocou tohto rozdelenia a ekvivalentných úprav rovnice je možné pred súčet nepárnych prvkov vyňať exponenciálny člen, nazývaný otáčací činiteľ alebo twiddle factor

$$W_N^m = e^{-j*2\pi*m/N}, \tag{3.2}$$

ktorý nie je závislý na indexe vstupnej vzorky. Každá polovica vstupného vektoru môže byť podobným spôsobom rozdelená a upravená a delenie pokračuje, pokiaľ nie je vstupná postupnosť rozdelená do dvojíc [9]. Jednotlivé otáčacie činitele pritom môžu byť vypočítané dopredu ako konštanty a tak je zmenšený počet operácií. Keďže sa jedná o komplexnú exponenciálnu funkciu, sú jej hodnoty súmerné podľa reálnej osi Gaussovej roviny a počet konštánt tak môže byť zredukovaný [13] .

Výsledné rovnice FFT algoritmov bývajú zvyčajne zobrazované v tzv. motýlikovom diagrame (z anglického názvu butterfly diagram [9]). Jeho príklad môžeme vidieť pre počet vzoriek *N* = 4 na obr. 3.1. Pre čo najprehľadnejšie zobrazenie bolo vykonaných niekoľko úprav. Každému samostatnému motýliku bol priradený jeho otáčací činiteľ (bola využitá možnosť výpočtu tejto konštanty z ostatných otáčacích činiteľov [13]), ktorým je nepárna vzorka násobená pred sčítaním aj odčítaním. Tento variant je optimalizovanou formou motýlika štruktúry DIT – z anglického Decimation-in-time, teda decimácia v čase [9]. Druhým typom štruktúry je frekvenčná decimácia (DIF), kedy sa otáčací činiteľ v optimalizovanej forme násobí len pre výsledok odčítania [9].

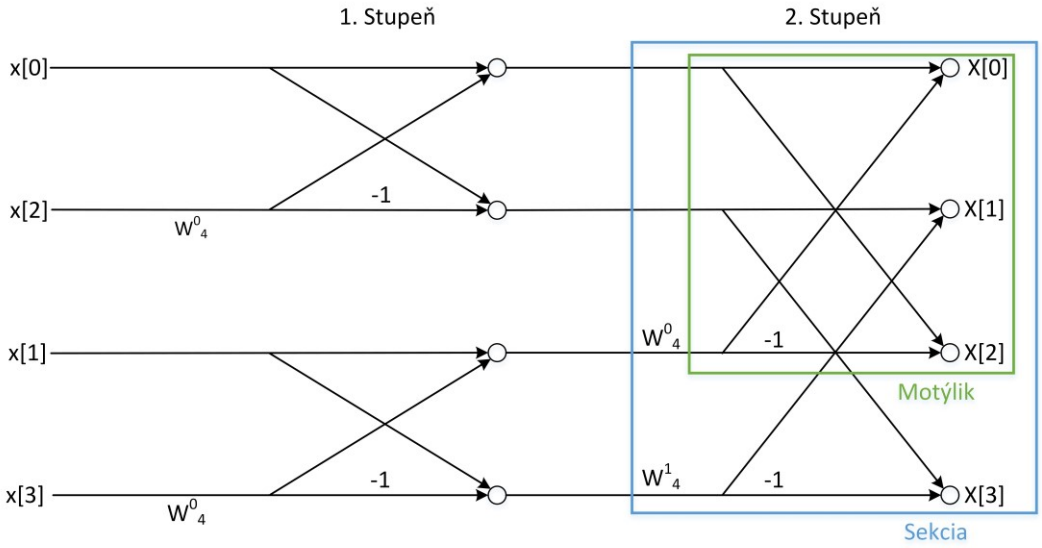
Aby boli výstupy FFT v prirodzenom poradí a diagram viac čitateľný, boli indexy vstupného vektoru pozmenené. Jedná sa o tzv. bitovú reverziu, kedy sa binárna hodnota indexu doslova „prevráti“ naopak [9]. Pre názornejší príklad uvádzame v rovnici 3.3 bitovú reverziu pre vytvorený motýlikový diagram z obr. 3.1

$$\begin{aligned} &N = 4, \\ &n = 0 = (00)_2 \rightarrow (00)_2 = 0, \\ &n = 1 = (01)_2 \rightarrow (10)_2 = 2, \\ &n = 2 = (10)_2 \rightarrow (01)_2 = 1, \\ &n = 3 = (11)_2 \rightarrow (11)_2 = 3. \end{aligned} \tag{3.3}$$

Na obr. 3.1 je tiež môžné vidieť rozdelenie diagramu na stupne, sekcie a motýliky. Počet stupňov, sekcií v každom stupni a motýlikov v jednotlivých sekciách sa dá vypočítať vzťahmi [14]

$$\begin{aligned} &n_{stupňov} = \log_2 N, \\ &n_{sekcií} = 2^{n_{stupňov}-s}, \\ &n_{motýlikov} = 2^{s-1}, \end{aligned} \tag{3.4}$$

kde *N* znovu označuje veľkosť vstupného vektoru a *s* predstavuje konkrétny stupeň, pre ktorý sú hodnoty počítané. Tieto vzťahy budú neskôr využité pri kontrole jednotlivých slučiek v referenčnom algoritme.

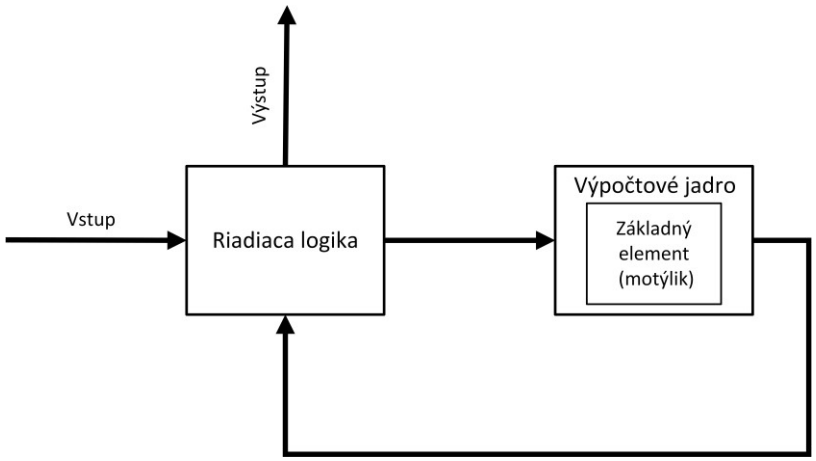


Obrázok 3.1 Motýlikový diagram algoritmu FFT štruktúry radix-2 DIT

3.2 Architektúry algoritmu FFT pre obvody FPGA a ASIC

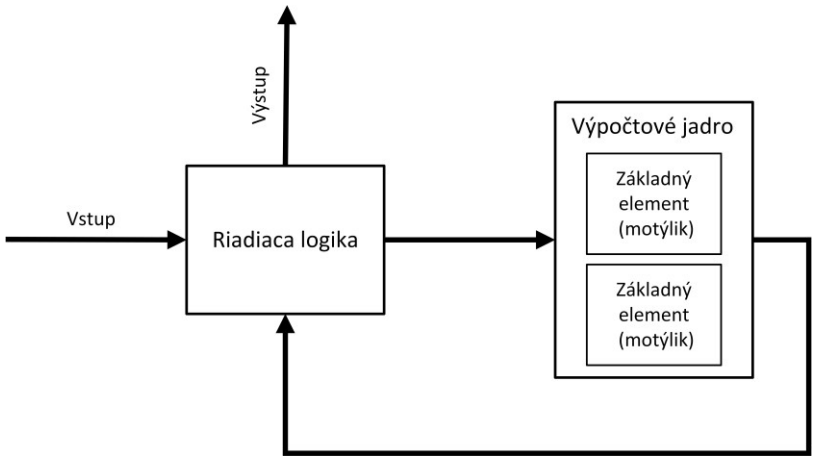
Pri návrhu akejkoľvek aplikácie pre obvody FPGA a ASIC sa kladie dôraz obzvlášť na plochu čipu. Okrem plochy sú ďalšími dôležitými parametrami tiež maximálna frekvencia, pri ktorej môže návrh pracovať, dátová priepustnosť alebo spotreba. S ohľadom na tieto skutočnosti je stále veľmi populárny pôvodný algoritmus Cooley – Tukey vo variantoch radix-2, radix-4, a tiež bol navrhnutý algoritmus pozostávajúci z kombinácií dvoch predchádzajúcich riešení, nazývaný splix-radix [15]. Rôzne typy architektúr budú predstavené na algoritme štruktúry radix-2, pričom tieto architektúry môžu byť analogicky aplikované na štruktúru radix-4.

V architektúrach s algoritmom radix-2 je základným výpočtovým blokom modul na výpočet motýlika dvoch vstupných vzoriek, ktorý môže byť navrhnutý pomocou násobičky, sčítačky a odčítačky komplexných čísel [12]. Prvý typ architektúry, ktorý bude predstavený, využíva princíp spätnej väzby, ako je možné vidieť na obr. 3.2. V tomto prípade je výpočtová časť najkompaktnejšia, keďže sa skladá iba z jedného modulu motýlika, ktorý získava vstupné vzorky z riadiacej logiky. Vypočítané hodnoty sú následne spätne zaslané riadiacej logike. Tá má teda na starosti manipuláciu so vstupnými dátami aj výsledkami z výpočtovej časti. Pri optimálnom návrhu sa môžu registre v riadiacej logike využívať veľmi efektívne a môžu slúžiť pre úschovu vstupnej aj výstupnej postupnosti.



Obrázok 3.2 Architektúra s jednoduchým výpočtovým jadrom a spätnou väzbou

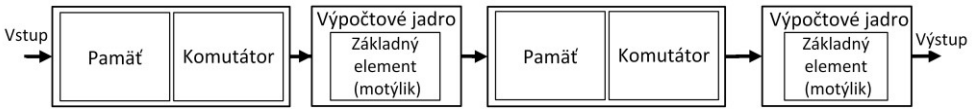
Pokiaľ záleží na kratšej latencii obvodu, teda rýchlejšom získaní výstupnej postupnosti, môže byť výpočet paralelizovaný väčším počtom motýlikov vo výpočtovom jadre. Konkrétny príklad pre počet vzoriek $N = 4$ je znázornený na obr. 3.3. Riadiaca logika v tomto prípade pripraví vstupy na spracovanie pre celý jeden stupeň výpočtu. Pri porovnaní s prvou spomínanou architektúrou pri rovnakom počte vstupných vzoriek sa tak vo výpočtovom jadre nachádzajú dva elementárne motýliky a sú potrebné dve iterácie výpočtu. Výpočtové jadro z obr. 3.2 musí svoje výpočty opakovať štyrikrát. Nevýhodou tejto realizácie výpočtu môže byť veľký nárast plochy výsledného obvodu pri väčšom počte vstupných vzoriek *N*.



Obrázok 3.3 Architektúra s postupným výpočtom stupňov a spätnou väzbou pre N = 4

Dátová priepustnosť môže byť ďalším parametrom, na ktorý sa pri návrhu architektúry môže klásť dôraz. Pre vyššiu priepustnosť dát sa využíva reťazové spracovávanie (z anglického slova pipelining). Príklad takejto architektúry sa nachádza na obr. 3.4. Podľa počtu stupňov sa za sebou nachádzajú dvojice modulov riadiacej logiky a elementárneho motýlika, pričom komutátorové bloky vnútri riadiacej logiky by mali zabezpečiť vhodné vstupy z pamäte do motýlika pomocou správneho nastavenia latencie

jednotlivých medzivýsledkov [16].



Obrázok 3.4 Architektúra s využitím pipeliningu pre N = 4

3.3 Referenčný model FFT v jazyku C++

Vzhľadom na následnú implementáciu algoritmu FFT pomocou knižnice SystemC bol použitý pre referenčný model jazyk C++ kvôli jednoduchej migrácii a potrebným úpravám. Zároveň tento model slúži na overenie správnosti výpočtu. Zdrojový kód tohto modelu sa nachádza v prílohe A.1.

Ako základ pre tento model bol zvolený algoritmus typu radix-2 a samotný kód je možné rozdeliť na štyri základné časti. V prvej z nich sa generuje vektor vstupných hodnôt typu *double* v rozmedzí od -5 po 5, na ktorých bude vykonaná FFT. Vygenerované vzorky sú tiež uložené do súboru *input.txt* pre kontrolu vstavaným výpočtom v programe Matlab. Pri vstupnom vektore sa predpokladajú iba reálne hodnoty, preto je imaginárna časť vyplnená nulami funkciou *calloc*. Z preddefinovanej konštanty NUM_SAMPLES je tiež vypočítaný počet stupňov podľa rovnice 3.4.

Vstupný vektor je ďalej vo *for* slučke umiestnený do poľa s bitovo reverzovanými indexami. Bitová reverzia bola docielená manuálne princípom podobným prevodu čísla na binárnu podobu pomocou operácie modulo, ale s opačným priradením hodnoty bitov (z LSB sa stane MSB, atď.).

Tretia časť sa skladá z troch *for* cyklov, ktoré postupne prechádzajú každým základným motýlikom v jednotlivých stupňoch a sekciách. Počet týchto úrovní, a teda aj koniec korešpondujúcich cyklov, kontrolujú premenné *stages_cnt*, *section_cnt* a *bfly_cnt*, ktorých hodnoty sú dané vzťahmi z rovnice (3.4)

Pre každý motýlik sa vypočíta príslušný otáčací činiteľ, indexy poľa, s ktorými sa počíta, a spoločné časti výsledku pre reálnu a imaginárnu časť oboch výstupov motýlika podľa všeobecného vyjadrenia

$$\begin{aligned}
&X = (a + jb) + (x + jy) * (c + jd),\\
&Y = (a + jb) - (x + jy) * (c + jd),\\
&X = a + jb + xc + jdx + jcy - dy,\\
&Y = a + jb - xc - jdx - jcy + dy,\\
&X = (a + xc - dy) + j(b + dx + cy),\\
&Y = (a - xc + dy) + j(b - dx - dy),\\
&common_{res_{real}} = xc - dy,\\
&common_{res_{im}} = dx + cy,
\end{aligned} \tag{3.5}$$

v ktorom sú *X* a *Y* všeobecné komplexné výstupy motýlika, výrazy *a + jb*, *c + jd* sú všeobecnými vstupmi a *x + jy* je otáčacím činiteľom. Z vyjadrení 3.5 vyplývajú

spoločné časti výsledkov pre reálnu a imaginárnu časť, ktorú treba ku párnemu vstupu buď pričítať alebo odčítať. Následne sú výsledky uložené do toho istého poľa, z ktorého boli získané vstupné hodnoty pre motýlik.

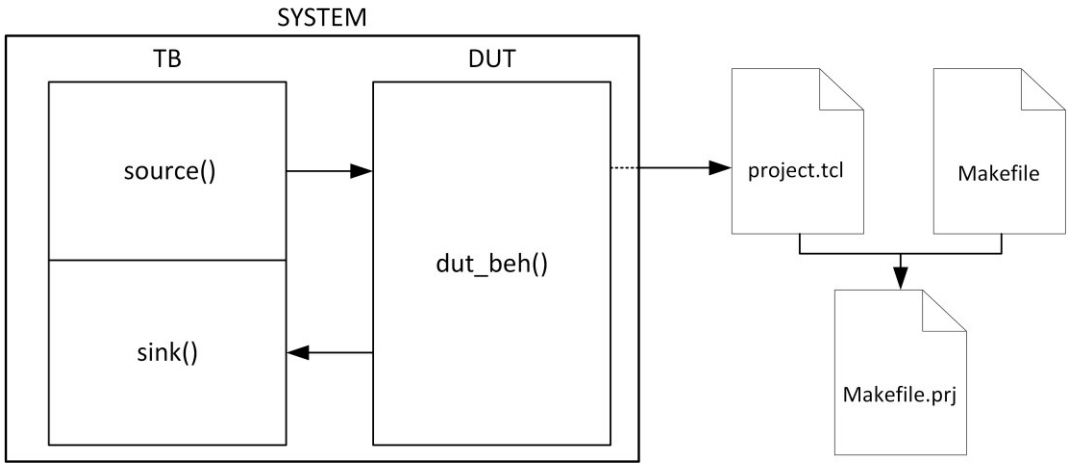
Na záver sú výsledky uložené do súboru *results.txt*, ktorý je porovnaný s výpočtom funkcie *fft* v programe Matlab. Zdrojový kód určený pre Matlab sa nachádza v prílohe A.2 a jeho úlohou je získať vygenerované vstupné hodnoty zo súboru *input.txt*, vypočítať FFT so svojou zabudovanou funkciou *fft* a vypočítať odchýlku medzi dvomi spôsobmi výpočtu. Ideálne by bol rozdiel všetkých zložiek spektra nulový, vzhľadom na odlišnú implementáciu je tento rozdiel pri počte vzoriek $N = 1024$ v rádoch maximálne 10^{-13}.

4 STRATUS HIGH-LEVEL SYNTHESIS

Nástroj Stratus High-Level Synthesis (ďalej označovaný skrátene Stratus) od firmy Cadence patrí do skupiny programov slúžiacich na behaviorálnu syntézu. Vstupom pre Stratus môže byť algoritmus napísaný pomocou jazyka C, C++ alebo SystemC. Z behaviorálneho kódu dokáže Stratus vygenerovať rozdielne RTL popisy v jazykoch Verilog alebo SystemC na základe odlišných nastavení syntetizátora. Samotným výstup je v podobe stavového automatu s dátovou cestou. S týmito výstupmi je možné naďalej pracovať priamo v prostredí Stratusu, kde môže byť RTL kód analyzovaný, funkčne overený a tiež prevedený na hradlovú úroveň, pokiaľ je dostupný podporovaný nástroj na logickú syntézu [17].

4.1 Štruktúra projektu v programe Stratus

Pri založení nového projektu Stratus najskôr vytvorí všetky súbory potrebné pre začiatok tvorby návrhu. Predpripravený projekt tak má hierarchiu znázornenú na obr. 4.1.



Obrázok 4.1 Štruktúra projektu v nástroji Stratus

V ľavej časti obr. 4.1 sa nachádza hlavný modul *SYSTEM*, knižnice SystemC s pomocou ktorého sa vykonávajú simulácie navrhovaného obvodu (modul *DUT*) na všetkých úrovniach abstrakcie. Tento modul má za úlohu vytvoriť inštancie nižších modulov, teda *TB* (z anglického slova testbench) a *DUT* a navzájom ich prepojiť. V rámci hlavného modulu je tiež definovaný hodinový signál a reset. Modul *TB* sa obvykle skladá z dvoch hlavných funkcií – *source()* a *sink()*. Prvá spomenutá funkcia zapisuje vstupné stimuly do *DUT* a druhá následne získava výstupné hodnoty [18].

Moduly *SYSTEM* a *TB* nie sú určené pre syntézu, slúžia iba na overenie funkčnosti návrhu. Z toho dôvodu bývajú označované ako systémové moduly. Definície všetkých modulov SystemC spolu s ich konštruktormi sa pritom väčšinou nachádzajú v hlavičkových súboroch s príponou .h, a telá funkcií v zdrojovom súbore s príponami .cpp. Samotná simulácia je spustená aj ukončená v tele funkcie *main()* [18].

4.1.1 Možnosti prepojenia modulov

Samotný princíp prepojenia modulov už bol predstavený v kapitole 2.1.2. Aby však výmena dát medzi nimi prebehla v poriadku, je potrebné, aby si moduly medzi sebou určitým spôsobom oznámili, že sú na danú komunikáciu pripravené. Jedná sa jednak o situáciu na vstupoch – teda či je modul pripravený prijať nové dáta, ale aj na výstupoch – či je schopný vysunúť výstupné dáta ďalšiemu modulu. Z týchto dôvodov sa implementuje tzv. handshake protokol, ktorým sa potvrdzuje tok dát [18].

Návrhár ma v programe Stratus pri voľbe prepojenia viac možností. Môže si navrhnúť svoj vlastný protokol, alebo môže použiť predpripravené triedy rozhrania *cynw_p2p* od Stratusu. Toto komunikačné rozhranie je plno syntetizovateľné a má v sebe implementovaný svoj vlastný handshake protokol. Časť názvu rozhrania „p2p“ indikuje prepojenie z bodu A do bodu B, ale je ho možné využiť aj pre paralelné porty [18]. Rozhranie *cynw_p2p* v sebe ukrýva množstvo metód a možností, jeho podrobnejšie skúmanie ale nie je primárnym účelom tejto práce.

4.1.2 Projektový súbor *project.tcl* a *Makefile*

Spôsob, akým bude samotný navrhovaný modul implementovaný do RTL podoby, ovládajú ďalšie súbory obsahujúce príslušné direktívy a príkazy pre nástroj Stratus. Táto časť projektu je na obr. 4.1 zobrazená v pravej časti. Jedným z týchto súborov je hlavný projektový súbor *project.tcl* definujúci všetky hlavné nastavenia a parametre projektu [18].

Nevyhnutnou súčasťou hlavného projektového súboru je definovanie technologickej knižnice súčiastok. Táto knižnica máva príponu .lib a obsahuje informácie od výrobcu IO o jednotlivých súčiastkach a ich parametroch, ako je napr. časovanie, plocha a spotreba. V prípade, že pre určitú časť kódu neexistuje korešpondujúca súčiastka, súčasť Stratusu s názvom CellMathDesigner sa pokúsi vytvoriť vhodnú alternatívu počas behu syntézy [18].

Okrem povinnej technologickej knižnice môžu byť pri projekte využité aj ďalšie typy knižníc. Patrí k nim aj pamäťová knižnica a knižnica vytvorená súčasťou programu Stratus s názvom Interface Generator, pomocou ktorého je možné vygenerovať prispôsobiteľné rozhranie v rámci dizajnu, ako je lineárny buffer, cirkulárny buffer a. i. Posledným druhom knižnice je knižnica vytvorených komponentov, ktoré sa do nej ukladajú počas behu behaviorálnej syntézy, ktorá tak môže pri ďalších spusteniach prebehnúť rýchlejšie [18].

Medzi ďalšie potrebné nastavenia projektu patrí definícia hodinového signálu, systémových modulov a modulov určených pre syntézy a simulácie. Pre každý syntetizovaný modul – *hls_module* môže byť vytvorených viacero konfigurácií s rôznymi nastaveniami atribútov syntézy, označovaných ako *hls_config*, simulačné konfigurácie na rôznych úrovniach abstrakcie – *sim_config*, a konfigurácie pre logickú syntézu – *logic_synthesis_config*. Jednoduchý projektový súbor *project.tcl* by mohol vyzerať napríklad takto:

```
# Project file
# Technology Library
set LIB_PATH "./tech_library.lib"
```

```
use_tech_lib "$LIB_PATH"
# Clock for each tool (in units from technology library - ns)
set_attr cc_options " -DCLOCK_PERIOD=10.0"
set_attr hls_cc_options " -DCLOCK_PERIOD=10.0"
set_attr clock_period 10
# System modules (not synthesizable)
define_system_module main main.cpp
define_system_module system system.cpp
define_system_module tb tb.cpp
# hls_module definition
define_hls_module dut dut.cpp
# hls_config definitions  for hls_module dut
define_hls_config dut BASIC
define_hls_config dut DPA
# logic_synthesis_config definiton for each hls_config
define_logic_synthesis_config GATES_BASIC {dut BASIC}
define_logic_synthesis_config GATES_DPA {dut DPA}
# sim_config - behavioral and RTL simulation for each hls_config
define_sim_config BEHAVIORAL {dut}
define_sim_config V_BASIC {dut RTL_V BASIC}
define_sim_config V_DPA {dut RTL_V DPA}
```

Okrem vyššie spomenutých nastavení je v *project.tcl* možné nastaviť množstvo iných parametrov týkajúcich sa aj integrácie ďalších nástrojov v HLS metodike návrhu s použitím programu Stratus. Existuje tak možnosť voľby simulátora Verilog a SystemC, nástroja pre logickú syntézu, programu na zobrazenie priebehov signálov atď.

Z *project.tcl* súboru je následne vygenerovaný tzv. *Makefile.prj*, ktorý automatizuje beh programu Stratus a umožňuje vykonávať rozličné úlohy a príkazy v rámci daného projektu. *Makefile.prj* tiež umožňuje predať argumenty ďalším nástrojom, ktoré majú vykonať požadovanú úlohu, napr. kompilátoru jazyka C++. Pre možnosť vkladania týchto úprav je nástrojom Stratus vygenerovaný hlavný *Makefile* súbor, v ktorom je zahrnutý vygenerovaný *Makefile.prj* s pomocou *#include* direktívy [18].

4.2 Odlišné prístupy k návrhu z časového hľadiska

Nástroj Stratus je schopný vytvoriť z čisto algoritmického zápisu bez časovania RTL popis s časovým rozvrhnutím jednotlivých operácií. Napriek tejto vlastnosti môžu nastať situácie, kedy návrhár potrebuje presne rozvrhnúť, v ktorom cykle sa vykoná daná operácia. Ako už bolo spomenuté v kapitole 4.1.1, jedným z takých prípadov je komunikácia medzi jednotlivými SystemC modulmi. Kvôli týmto odlišným prístupom je preto možné navrhovať časti kódu dvomi spôsobmi, a to buď ako fixne časované bloky, alebo časovo nerozplánované bloky. Každý blok kódu sa pritom oddeľuje zloženými zátvorkami [18].

Fixne časované bloky navrhuje z časového hľadiska iba návrhár. Stratus teda do týchto častí kódu nezasahuje, v prípade nesprávneho časovania je len nahlásená chyba. Takýto blok sa musí hneď po otvorenej zloženej zátvorke označiť direktívou *HLS_DEFINE_PROTOCOL* a časti kódu sa oddeľujú funkciou *wait()*, teda hodinovými cyklami. Funkciou *wait()* sa dá pri tomto štýle jednoducho ovplyvniť, či budú po vstupoch nasledovať registre – stačí ju pridať po čítaní vstupov. Je dobré poznamenať, že v prípade takýchto častí kódu si návrhár priamo špecifikuje podobu samotného výstupného stavového automatu [18].

Na rozdiel od fixne časovaných blokov má Stratus možnosť optimalizovať bloky časovo nerozplánované tak, aby dosiahol požadované parametre a atribúty návrhu. Pokiaľ nie je nutné pridávanie vstupných registrov, je možné tak nastaviť za pomoci direktívy *HLS_ASSUME_STABLE* [18].

4.3 Ovplyvnenie výsledných parametrov implementácie

Výsledný RTL kód vygenerovaný Stratusom môže byť pozmenený dvomi odlišnými spôsobmi. Prvým z nich je optimalizácia samotného vstupného kódu, ktorého podoba môže vplývať na výsledok syntézy. Základom transformácie vstupného algoritmu je konverzia dátových typov jazyka C na dátové typy z knižnice SystemC – *sc_int* a *sc_uint*. V prípade vyšších bitových šírok je možné použiť dátové typy *sc_bigint* a *sc_biguint*.

Druhou možnosťou je nastavenie požadovaných parametrov a atribútov ovplyvňujúcich výsledky syntézy, pričom samotný atribút môže byť aplikovaný lokálne – na blok kódu, poprípade na premennú, alebo globálne. Ak sa jedná o blok kódu, mala by sa direktíva nachádzať na začiatku bloku. Pre premennú zasa platí, že direktíva by mala byť umiestnená v oblasti pôsobnosti premennej. Atribút možno nastaviť globálne v *project.tcl* súbore buď pre celý projekt, alebo pre konkrétny *hls_config*. Pre nastavenie týchto parametrov sa pritom využije príkaz *set_attr*. V prípade odlišných nastavení má vyššiu prioritu lokálna direktíva [18].

4.3.1 Časovanie

Stratus umožňuje ovplyvniť agresivitu časovania s nastavením parametru *timing_aggresion* v *project.tcl* súbore. Tento parameter môže nadobúdať celočíselné hodnoty od -10 po 10, poprípade môže mať hodnotu *off*. Vyššia hodnota parametra spôsobí, že Stratus vloží medzi registre viac obvodov a pre nástroj na logickú syntézu bude potom náročnejšie splniť požadované časové parametre. Nižšia agresivita zas vytvorí konzervatívnejší návrh časovania, môže však viesť k vyššej latencii a väčšiemu množstvu registrov, a teda aj celkovej plochy návrhu. Hodnota parametra *timing_aggresion* rovná 0 je doporučená Stratusom ako dobrý začiatok pri návrhu, pretože poskytuje dobrý celkový manažment časovania [19].

Od hodnoty 0 sa líši nastavenie *off*, ktoré inštruuje využitie všetkých predvolených nastavení v rámci časovania. Parameter *timing_aggresion* je teda akýmsi súhrnom viacerých nastaviteľných atribútov, ktoré je v prípade potreby možné meniť aj samostatne. Zahŕňa v sebe parameter *cycle_slack* a *path_delay_limit*. Direktíva *cycle_slack* nastavuje nový čas, s ktorým Stratus bude počítať pri návrhu časovania. Pokiaľ je napríklad *cycle_slack* 2 ns a perióda hodín je 10 ns, Stratus bude brať do úvahy pri výpočtoch ich rozdiel, teda 8 ns. Pokiaľ bude pri niektorých súčiastkach tento čas prekročený, bude automaticky vložený ďalší hodinový cyklus. Parameter *path_delay_limit* zas slúži na špecifikáciu najdlhšej akceptovateľnej dĺžky dátovej cesty v percentách hodinového signálu. Oba tieto parametre je možné meniť samostatne, pokiaľ to ale nie je nutné, Stratus odporúča využitie súhrnného nastavenia *timing_aggresion* [19].

Stratus zvykne často vytvoriť RTL výstup, ktorého predpokladané oneskorenie je dlhšie ako špecifikovaná hodnota hodinového signálu. Je to z toho dôvodu, že nástroj na

logickú syntézu dokáže neskôr tieto oneskorenia vykompenzovať [18].

4.3.2 Latencia a plánovanie

Latenciu možno definovať ako počet hodinových cyklov, za ktorú obvod vyprodukuje výstup. Tento parameter je možné nastaviť pre jednotlivé bloky kódu direktívou *HLS_CONSTRAIN_LATENCY*, kde sa dá udať minimálny a maximálny počet hodinových cyklov. Pre nelimitovanú maximálnu latenciu je možné udať druhý parameter ako -1. V tom prípade sa Stratus pokúsi čo najviac minimalizovať plochu obvodu [19].

Plánovací algoritmus Stratusu je predvolene určený pre dizajny s minimom kontrolnej logiky a väčším podielom dátových ciest. Pokiaľ dizajn obsahuje veľké množstvo kontrolnej logiky, zlepšenie latencie sa môže docieliť zapnutím agresívneho plánovania pomocou direktív *sched_aggresive_1* alebo *sched_aggresive_2*. Nastavenie *sched_aggresive_1* vytvorí z ovládacích výrazov dátové cesty a dokáže umiestniť ovládacie súčiastky paralelne pre zníženie latencie. Direktíva *sched_aggresive_2* slúži na dodatočné pokúšanie sa o zníženie plochy a môže viesť k dlhšiemu procesu syntézy. Direktívu *sched_aggresive_1* je tiež možné ovládať aj lokálne pre konkrétny blok kódu [19].

4.3.3 Slučky

Nástroj Stratus dáva používateľovi kontrolu nad tým, akým spôsobom bude každá slučka interpretovaná. Hlavnou operáciou so slučkou je jej rozvinutie, ktoré ovplyvňuje, či je logika vnútri slučky replikovaná, alebo nie. Predvolene táto možnosť nastavená nie je, okrem 2 výnimiek - reťazených slučiek a slučiek vytvorených Datapath Optimizerom (druhá možnosť je vysvetlená v kapitole 4.3.4). Pre nerozvinutú slučku platí, že pre výpočet vnútri nej sú použité stále tie isté hardvérové prostriedky, čo znamená väčšiu latenciu potrebnú pre dokončenie slučky [18].

Rozvíjanie slučiek je možné nastaviť globálne, alebo je možné každú slučku upraviť lokálnym nastavením. Pri lokálnych nastaveniach je možné rozvíjanie viac špecifikovať. Rozvinúť sa dá len konkrétna, alebo aj všetky vnútorné slučky, okrem toho môže byť nastavené čiastočné rozvinutie slučky podľa zvolenej hodnoty, alebo iné možnosti. Je dôležité poznamenať, že aby slučka mohla byť rozvinutá, musí Stratus dopredu vedieť konečný počet iterácií [18].

Ďalším variantom úpravy interpretácie slučiek je jej reťazenie, alebo pipelining. Táto možnosť umožňuje začať jednu iteráciu slučky pred koncom tej predošlej, a tak zvýšiť priepustnosť dizajnu. Aby však reťazové spracovanie bolo možné, musí byť kód správne napísaný. Hlavnou podmienkou je, aby daná slučka a aj všetky vnútorné slučky boli kompletne rozvinuté. Stratus túto operáciu koná automaticky, musí však byť dopredu daný konečný počet iterácií. Chyby môže tiež vyprodukovať závislosť dát v slučke, nesprávne navrhnuté fixne časované bloky, konflikty pri prístupoch k portom, alebo nesprávne navrhnutá podmienka pre ukončenie slučky [18].

Pokiaľ sa nedá zaručiť, že ďalšie bloky budú schopné predať alebo prijať nové dáta, musí byť reťazená slučka schopná pozastaviť svoju funkciu. Toto pozastavenie môže nastať napr. na vstupe, keď je blok pripravený prijať dáta, ale predošlý mu nie je schopný ich poskytnúť. Opačným prípadom je pozastavenie na výstupe, kedy daný blok môže

predať nové dáta, ale nasledujúci ich nedokáže prijať. Treťou možnosťou je, že podnet na pozastavenie môže vydať externý signál. Tieto tri typy môžu byť implementované dvomi spôsobmi. Buď pri podnete prestáva pracovať celý blok, alebo je pozastavená iba určená časť, zvyšok bloku pokračuje ďalej. Stratus dokáže implementovať všetky tieto kombinácie okrem pozastavenia určitej časti na výstupe [18].

4.3.4 Dátové cesty

V niektorých prípadoch dokáže Stratus dosiahnuť lepšie výsledky syntézy za pomoci jeho časti s názvom Datapath Optimizer (skrátene DpOpt). Ten je integrovaný v jadre syntetizátora a vytvára nové súčiastky s použitím ďalšieho podprogramu CellMath Designer, ktoré sa stávajú súčasťou knižnice vytvorených komponentov, a tak môžu byť znovu použité v dizajne. Využitie DpOpt môže byť špecifikované globálne, alebo pre samostatné bloky v dizajne [18].

Pre zníženie spotreby je možné pridať povoľovací vstup do súčiastky DpOpt, aby bola aktívna len v tom prípade, keď sú platné dátové vstupy. Túto možnosť však Stratus povoľuje iba v prípade vyššej licencie [18].

4.3.5 Výrazy

Okrem zmeny štýlu písania kódu môže byť kvalita výsledkov ovplyvnená optimalizáciou výrazov v dizajne. Jednou z možností je eliminácia spoločných výrazov *comm_subexp_elim*. Pri povolenom atribúte Stratus deteguje, či je niektorý výraz ďalej použitý v kóde viackrát. Pokiaľ nájde takýto prípad, zdieľa prvý výsledok v ďalších operáciách [19].

Iným variantom je parameter optimalizácie výrazov *balance_expr*, ktorý môže nadobúdať tri hodnoty. S hodnotou *width* sa Stratus pokúša minimalizovať bitové šírky vo všetkých výrazov, a teda vytvárať čo najmenšie komponenty. Okrem hodnoty *width* môže mať ešte hodnotu *delay*, pri ktorej kladie dôraz na čo najmenšie oneskorenie každého výrazu, alebo hodnotu *off*, kedy k žiadnej z týchto optimalizácií nedochádza. Tento globálne nastavený parameter môže byť nahradený lokálnou direktívou *HLS_BALANCE_EXPR* [19].

4.3.6 Práca s poľami a pamäťami

Úložisko môže byť v návrhu implementované dvomi spôsobmi – implicitne alebo explicitne. Pri implicitnej implementácii sa v kóde nachádzajú klasické polia C++, ktoré sú mapované do hardvérového zdroja, v explicitnom spôsobe sú vytvorené inštancie úložiska v zdrojovom kóde. Táto kapitola sa venuje čisto implicitnému prístupu [18].

Stratus podporuje všetky typy polí jazyka C++, a tieto polia môžu byť syntetizované do troch hardvérových architektúr – pamäte ROM, banky registrov, alebo môže byť každý element poľa samostatnou premennou. Predvolene mapuje Stratus polia do pamätí, pričom sa vždy snaží využiť súčiastku z definovanej knižnice s čo najmenšími bitovými šírkami. Mapovanie poľa do banky registrov je ovládané lokálnou direktívou, a u tejto možnosti je čítanie asynchrónne, zápis trvá jeden cyklus a pokiaľ nie sú príslušné indexy v rozpore, je povolený súbežný prístup [18].

Pre poslednú možnosť je potrebné využitie globálnej direktívy *flatten_arrays*

(existuje aj lokálny variant). Výhodou využitia tejto možnosti je paralelný prístup k jednotlivým elementom, na druhej strane to môže znamenať nárast plochy a zvýšenú komplexnosť návrhu. Táto direktíva sploštenia polí je však veľmi efektívna v kombinácii s rozvinutím slučiek, bez neho je efekt rozvinutia totižto menej badateľný [19].

Pokiaľ sú určité časti konštánt rovnaké, je možné dosiahnuť úsporu plochy direktívou *inline_partial_constants*. S touto direktívou sú tak dané spoločné časti privedené na jednu referenciu [19].

5 BEHAVIORÁLNA SYNTÉZA ALGORITMU FFT

Po zoznámení sa s možnosťami nástroja Stratus je možné ho použiť na prevod behaviorálneho algoritmu FFT z Prílohy A.1 do RTL podoby. Aby bol však tento proces uskutočniteľný, je potrebné vykonať niekoľko modifikácií. Jednak ide o definovanie hlavného navrhovaného SystemC modulu a jeho rozhrania, ale hlavne o úpravy v rámci samotného behaviorálneho popisu algoritmu FFT. Následne môže byť vybraná vhodná kombinácia parametrov syntézy a syntetizované rozličné architektúry toho istého vstupného algoritmu. Všetky syntézy prebehli s použitím knižnice *gscl45nm.lib* a funkčnosť návrhov bola otestovaná podľa metodiky popísanej v kapitole 5.4.

5.1 Modifikácia referenčného algoritmu

V prvom rade musí byť dátový typ *float* v návrhu reprezentovaný syntetizovateľným dátovým typom z knižnice SystemC. Na modelovanie operácií s desatinnými číslami s pevnou desatinnou čiarkou sa ako prvé logicky ponúkajú dátové typy *sc_fixed* a *sc_ufixed*. Avšak tieto dátové typy momentálne nie sú nástrojom Stratus pre syntézu podporované, a tak musí byť desatinné číslo určitým spôsobom vyjadrené za pomoci celočíselných dátových typov.

Pre implementáciu desatinných čísel bol zvolený dátový typ *sc_int* so spoločnou šírkou všetkých elementov nachádzajúcich sa v dizajne – 20 bitov. Zo zvoleného rozsahu bolo 10 bitov vyhradených pre desatinnú časť, a keďže MSB tohto dátového typu slúži ako znamienkový bit, pre celú časť čísla tak ostalo 9 bitov. Takto zvolená bitová šírka by mala poskytovať dostatočnú presnosť a rozsah a v prípade potreby je možné túto šírku jednoducho upraviť.

Pre prevod čísla dátového typu *float* na takto konštruovaný dátový typ *sc_int* ho stačí vynásobiť číslom 2 umocneným počtom bitov, o ktoré bola posunutá desatinná čiarka, alebo vykonať bitový posun – tieto operácie sú ekvivalentné. Nevyhnutné ošetrenie vyžaduje operácia násobenia, kedy musia byť oba činitele upravené na bitovú šírku výsledku operácie – teda na dvojnásobok svojej pôvodnej šírky. Výsledok operácie musí byť následne znova bitovo posunutý o počet desatinných miest a znovu je mu prinavrátená pôvodná bitová šírka. Celú operáciu násobenia demonštruje všeobecné vyjadrenie

$$\begin{aligned} &\mathrm{sc_{int}} < \mathrm{BW} > \mathrm{z} = (\mathrm{sc_{int}} < \mathrm{BW} >) \\ &\big(\big((\mathrm{sc_{int}} < 2 * \mathrm{BW} >)(\mathrm{x}) * (\mathrm{sc_{int}} < 2 * \mathrm{BW} >)(\mathrm{y})\big) \gg \mathrm{DPW}\big), \end{aligned} \tag{5.1}$$

kde *BW* je celková bitová šírka, *DPW* je bitová šírka desatinnej časti, *z* je súčin a *x* a *y* sú činitele. Je vidieť, že jediná operácia násobenia môže vyzerať neprehľadne, z toho dôvodu boli rozdelené aj výpočty vo vnútri motýlika [20]. Ako bude ukázané v kapitole 5.2.5, pri vhodnom nastavení parametrov syntézy dokáže byť vplyv tejto zmeny na HLS potlačený.

Ďalšou zmenou v algoritme je výpočet konštánt – otáčacích činiteľov. Spôsob

výpočtu ostáva zachovaný, len sú jednotlivé otáčacie činitele uložené dopredu do polí *constants_real[NUM_SAMPLES/2]* a *constants_im[NUM_SAMPLES/2]* v slučke typu *for*. Na základe toho je možné usúdiť, že Stratus dokáže rozpoznať využitie funkcií *sin()* a *cos()* z knižnice *math.h,* a že sa hodnoty v poliach ďalej nemenia, preto ich vypočíta dopredu. Nie je tak nutné konštanty pripravovať iným spôsobom a importovať ich napríklad zo súboru. Konštanty sú vo všetkých prípadoch implementované ako samostatné elementy pomocou direktívy *flatten_arrays*, čo umožňuje paralelný prístup, no zároveň aj možné zvýšenie plochy. Variant s využitím pamätí ROM nie je odskúšaný, keďže nebola dostupná pamäťová knižnica.

Napriek tejto skutočnosti však Stratus vykazuje problém s funkciou *log2()* slúžiacou na výpočet stupňov pre daný počet vzoriek. Hodnota tejto premennej je tiež po celý čas konštantná, Stratus tentokrát ale výpočet dopredu neprevedie a snaží sa pre danú operáciu nájsť vhodnú komponentu. Z toho dôvodu bol výpočet logaritmu so základom 2 prepísaný, aby bol návrh vhodný pre viaceré počty vstupných vzoriek. Podobným prípadom je aj funkcia *pow2()*, ktorá je využitá vo viacerých častiach algoritmu. Jej zjednodušená verzia *pow_2()* je samostatnou funkciou v rámci SystemC modulu algoritmu FFT a je volaná v prípade potreby. Okrem týchto zmien ostalo jadro algoritmu v pôvodnej podobe a nachádza sa v prílohe B.2.

5.2 Vplyv nastavovaných atribútov na výsledok syntézy

Najskôr sa pozrieme na odlišné konfigurácie Stratusu pri tom istom počte vzoriek a vstupno-výstupnom rozhraní. Tak môže byť zistené, aký vplyv majú jednotlivé parametre na výstupnú RTL podobu návrhu, pričom ostatné parametre ostávajú nezmenené. Porovnanie bude vykonané so vstupným vektorom o veľkosti 32 vzoriek a ôsmimi vstupnými a výstupnými portami pre *hls_config* BASIC 1 (s obmedzením výpočtu jednej iterácie algoritmu na 1 cyklus).

5.2.1 Časovanie

Jednotlivé výsledky syntézy pre rozličné hodnoty parametra *timing_aggression* možno vidieť v tab. 5.1. Zo získaných údajov je badateľný priamy súvis *timing_aggression* s časovou hodnotou najdlhšej dátovej cesty, čo korešponduje s dokumentáciou Stratusu. Na druhej strane je však zaujímavý fakt, že podobnú závislosť nie je možné pozorovať pri odhade plochy na čipe. Stratus dokázal napríklad pri hodnote -10 dosiahnuť menšiu hodnotu najdlhšej dátovej cesty zároveň s menšou plochou oproti syntéze s hodnotou *timing_aggression* rovnou -5. Tento jav môže byť spôsobený väčším počtom iterácií behu programu, ktorý pri optimalizácii časovania dokáže viac optimalizovať aj plochu. Rovnako je badateľný rozdiel medzi hodnotou *off* oproti ostatným, kedy sa na časovanie nekladie žiaden dôraz. Výsledky syntézy sú však pri hodnote *off* stále lepšie ako pri vyšších hodnotách, ako je 5 a 10.

Aby bola už pri behu HLS splnená požiadavka pre periódu hodinového signálu, bola potrebná hodnota *timing_aggresion* rovná -3. Pri rovnakej ploche však Stratus dokázal vytvoriť omnoho agresívnejšie časovanie pri použití hodnoty -10. Kapitola 4.3.1 sa tiež zmieňuje o tom, že Stratus väčšinou vytvorí návrh s väčším časom najdlhšej dátovej cesty, pretože nástroj pre logickú syntézu dokáže túto hodnotu ešte znížiť. Overenie tohto

tvrdenia sa nachádza vo štvrtom stĺpci tab. 5.1, kde sa nachádzajú hodnoty najdlhšej dátovej cesty po procese mapovania. Pri nižších hodnotách *timing_aggression* parametra sú časové odhady nástroja HLS o niečo menšie, ako sú po mapovaní, od hodnoty nula je však vidieť potvrdenie predpokladu o znížení požadovanej hodnoty. Pri príliš vysokých hodnotách *timing_aggression* už ale nástroj pre logickú syntézu nebol schopný dosiahnuť požiadavku pre periódu hodinového signálu, z toho dôvodu je dobré nastaviť túto direktívu s určitou rezervou.

Tabuľka 5.1 Vplyv parametra *timing_aggression* na výsledky syntézy

timing_aggression	**Plocha [µm^2]**	**Najdlhšia dátová cesta po HLS [ns]**	**Najdlhšia dátová cesta po logickej syntéze [ns]**
off	146238	17,51	10,85
-10	177066	6,64	7,86
-5	203068	7,12	8,87
-3	177105	8,71	8,39
0	157452	12,33	8,07
5	149425	20,44	12,17
10	148678	24,55	12,75

5.2.2 Latencia a plánovanie

Reštrikcia pre latenciu je samotnou súčasťou *hls_config* BASIC 1, v ktorej má prebehnúť 1 iterácia algoritmu FFT za 1 hodinový cyklus. Pre porovnanie optimalizácie Stratusu bol preto vytvorený *hls_config* BASIC 2 nachádzajúci sa v treťom riadku tab. 5.2, v ktorom je toto obmedzenie znížené na 2 cykly. V poslednom riadku tab. 5.2 sa nachádza behaviorálna syntéza bez akéhokoľvek obmedzenia latencie. Práve z posledného riadku možno vydedukovať primárny cieľ optimalizácie Stratusu, ktorým je výsledná plocha návrhu na čipe. Pokiaľ Stratus nemá nijako obmedzenú hodnotu latencie, neberie na ňu ohľad a pokúša sa vytvoriť čo najmenšiu RTL podobu vstupného kódu, ktorá obsahuje minimálny počet komponent. V tomto prípade obsahuje dizajn iba 2 násobičky, 1 sčítačku, 1 odčítačku a menší počet registrov – 151. Tiež sa výrazne zmenšil čas najdlhšej dátovej cesty, a to na 7,39 ns. Výpočet pri takejto architektúre ale trvá až 139 cyklov.

Tabuľka 5.2 Vplyv obmedzenia latencie na výsledok syntézy

Obmedzenie latencie	**Plocha [µm^2]**	**Latencia [počet cyklov]**	**Najdlhšia dátová cesta [ns]**	**Počet násobičiek**	**Počet sčítačiek**	**Počet odčítačiek**	**Počet registrov**
sched_asap	323742	8	9,93	82	79	62	173
1	157452	9	12,33	24	45	41	217
2	138879	10	12,7	19	33	45	239
Bez obmedzenia	61130	139	7,39	2	1	1	151

Plánovanie je teoreticky možné ovplyvniť parametrami *sched_aggresive_1*

a *sched_aggresive_2*. Pri zapnutí týchto možností však žiaden rozdiel vo výstupnom RTL kóde nenastal. Nulový efekt týchto parametrov na výsledok syntézy však nemusí dokazovať, že sú úplne nepotrebné. Tento konkrétny návrh len nepotrebuje využiť väčšiu agresivitu pri rozplánovaní operácií. Okrem týchto dvoch parametrov obsahuje Stratus ešte ďalšiu možnosť s názvom *sched_asap*, ktorej patrí prvý riadok tab. 5.2. Pokiaľ je tento parameter zapnutý, Stratus by mal vytvoriť dizajn s čo najmenšou možnou latenciou. Pri využití tejto direktívy je pre testovaný hls_config latencia 8 hodinových cyklov, čo je o 1 cyklus menej ako klasický *hls_config* BASIC 1. Celková plocha návrhu ale narástla až na 323742 μm^2, čo je takmer dvojnásobok oproti návrhu s latenciou 9 cyklov. Stratus pri tejto direktíve teda neberie plochu vôbec do úvahy. Parameter *sched_asap* tak môže plniť informačnú funkciu o najmenšej dosiahnuteľnej latencii daného návrhu.

5.2.3 Slučky

Prvým odskúšaným nastavením je globálne rozvíjanie slučiek. Vypnutie tejto možnosti spôsobuje neúspešnú behaviorálnu syntézu z dôvodu nepovoleného výskytu desatinných čísel v návrhu. Jednotlivé konštanty sa v tomto prípade počítajú postupne a Stratus nedokáže identifikovať, že ich môže vypočítať dopredu. Syntéza ale neprebehla ani po pridaní lokálnej výnimky rozvinutia slučky s výpočtom konštánt. Pri obmedzení latencie výpočtu na 1 cyklus nie je Stratus schopný operácie vhodne rozplánovať, keďže pri zachovaní slučky v pôvodnom stave sa pre jednotlivé operácie využívajú rovnaké komponenty. Táto skutočnosť tak potvrdzuje potrebu rozvinutia slučiek v tomto návrhu.

Využitie reťazového spracovania z obr. 3.4 pri súčasnej podobe algoritmu nie je možné implementovať. Aby sa dalo pre danú slučku aplikovať reťazové spracovávanie, je potrebné dopredu vedieť konečný počet iterácií slučky. Podľa nástroja Stratus túto podmienku nespĺňa modifikovaná funkcia výpočtu druhej mocniny *pow_2()* a HLS tak nemohla prebehnúť.

5.2.4 Dátové cesty

Pre otestovanie možností Datapath Optimizeru v globálnej miere bol vytvorený samostatný *hls_config* DPA 1 a *hls_config* DPA 2, ktorých výsledky syntéz sú neskôr popísané v kapitole 5.3. Vytvorenie samostatnej komponenty za pomoci Datapath Optimizeru je tiež možné aj lokálne, a táto možnosť je využitá pre vytvorenie architektúr s využitím spätnej väzby a výpočtového bloku zloženého buď z jedného motýlika (obr. 3.2), alebo jedného stupňa (obr. 3.3). Napríklad blok kódu pre vytvorenie jedného motýlika môže byť realizovaný takýmto spôsobom:

```
{
BFLY HLS_DPOPT_REGION(NO_ALTS, "bfly", "bfly message");
//computation of FFT butterfly, NO_ALTS - ensures sharing of resources
}
```

Výsledky oboch behaviorálnych syntéz možno vidieť v tab. 5.3. V prípade využitia jedného motýlika latencia návrhu korešponduje s očakávanou hodnotou, keďže pri 32 vzorkách sa vo výpočte nachádza 80 dvojvstupových motýlikov podľa vzťahu:

$$n_{butterfly} = \log_2 N * \frac{N}{2} = \log_2 32 * \frac{32}{2} = 5 * 16 = 80, \quad (5.2)$$

kde N je počet vzoriek a N/2 je počet motýlikov v každom stupni, keďže je použitý

variant FFT radix-2. Nesedí ale plocha vytvoreného komponentu, ktorá by pre jeden FFT motýlik mala byť omnoho menšia. Opačný prípad nastáva pre *hls_config* ONE_STAGE. Plocha komponentu pre výpočet jedného stupňa je adekvátna vzhľadom na to, že by mala obsahovať 16 samostatných motýlikov. U tejto architektúry ale nesedí nesmierne veľká latencia 413 hodinových cyklov. Táto hodnota by sa mala pohybovať maximálne okolo čísla 10 pre 5 stupňov výpočtu. Z výsledkov v tab. 5.3 tak vyplýva, že pre vytvorenie lepšie optimalizovaných špecifických architektúr je potrebné viac uspôsobiť vstupný kód.

Tabuľka 5.3 Parametre vytvorených architektúr FFT so spätnou väzbou

hls_config	Celková plocha [μm^2]	Plocha DpOpt komponentu [μm^2]	Latencia [počet cyklov]
ONE_BFLY	78413	43410,7	87
ONE_STAGE	83614	45467,7	413

5.2.5 Výrazy

Spracovanie výrazov v algoritme je možné nastaviť viacerými parametrami, avšak na použitý algoritmus FFT má vplyv iba jeden, a to *comm_subexp_elim*. Pri jeho vypnutí dôjde ku zväčšeniu plochy z 157452 μm^2 na 161381 μm^2 a architektúra modulu je úplne pozmenená. Tento fakt je spôsobený hlavne z dôvodu rozdelenia výpočtov vnútri motýlika FFT na jednotlivé operácie z dôvodu prehľadnosti. Pri vypnutí tohto parametra tak nedochádza k zdieľaniu predošlých výsledkov a RTL popis má tým pádom väčšiu výslednú plochu návrhu na čipe. Nastavovanie parametrov *balance_expr* a *partial_inline_constant* nehrá v tomto návrhu žiadnu rolu, napriek tomu, že Stratus tieto parametre navrhuje vo výpise z HLS použiť.

5.3 Syntéza výslednej konfigurácie s odlišným vstupno-výstupným rozhraním

Na základe predošlých testov parametrov bol pre spoločnú behaviorálnu syntézu vybraný jednotný projektový súbor *project.tcl* nachádzajúci sa v prílohe B.9. Počty vstupných vzoriek boli zvolené s ohľadom na variant algoritmu FFT radix-2, pre ktorý platí, že táto hodnota musí byť druhou mocninou čísla 2. Pre ďalšie možné porovnanie optimalizácie vstupného kódu Stratusom boli zároveň vytvorené 4 samostatné projekty s modulmi FFT s odlišným vstupno-výstupným rozhraním. V prvom projekte sa na vstupe aj na výstupe nachádza sériová linka, vo zvyšných troch projektoch sa na vstup privádzajú paralelne 2, 4 alebo 8 vstupných vzoriek vektora. To isté platí aj pre výstupné rozhranie modulu. Ukážka projektu konkrétne pre 2 paralelné vstupy a výstupy je obsahom prílohy B.

Každá veľkosť algoritmu FFT v kombinácii s odlišným vstupno-výstupným rozhraním prešla behaviorálnou syntézou v štyroch základných konfiguráciách. Prvé dva tvoril *hls_config* BASIC, ktorý mal okrem spoločných nastavení lokálnou direktívou obmedzenú latenciu v hlavnej slučke výpočtu na 1 a 2 hodinové cykly. Lokálne direktívy boli samostatne umiestnené v súbore *directives.h* nachádzajúcom sa v prílohe B.3. Druhú dvojicu tvoril *hls_config* DPA s obdobnými obmedzeniami latencie, avšak s globálne nastaveným parametrom *dpopt_auto* na hodnotu *all*. DPA *hls_config* tak využíva

vstavaný algoritmus na optimalizáciu dátových ciest. Súhrnný prehľad všetkých vytvorených architektúr sa nachádza v prílohe C, v tejto kapitole budú samostatne zdôraznené pozorované zákonitosti vyplývajúce z výsledkov HLS.

Z výsledkov HLS možno usúdiť, že zásadný vplyv na výslednú architektúru pri všetkých konfiguráciách Stratusu má samotné vstupno-výstupné rozhranie. Vzhľadom na algoritmické vyjadrenie funkcie je pre výpočet dôležité, koľko vstupných vzoriek má modul od začiatku výpočtu k dispozícii. Táto závislosť je zobrazená v tab. 5.4 pre výber počtu vzoriek 16 a 64. Je spôsobená tým, že s väčším počtom vstupných hodnôt je možné behom jedného cyklu vykonať viac operácií a podľa toho uspôsobiť potrebný počet jednotlivých komponent. Väčší paralelizmus má vo všeobecnosti dopad na nižšiu latenciu výpočtu, ktorá je však vykúpená väčšou plochou kombinačnej časti a tiež väčším počtom registrov. Závislosť počtu registrov sa tiež odvíja aj od počtu vstupných vzoriek. Z tab. 5.4 je tento rozdiel zrejmý, kedy pri 16 vstupných vzorkách počet registrov s väčším paralelizmom jasne narastá, keďže je potrebné po každej iterácii držať väčší počet medzivýsledkov. Pri 64 vzorkách je už počet registrov pri 4 a 8 vstupno-výstupných rozhraniach takmer totožný (404 oproti 401 registrom).

Tabuľka 5.4 Vplyv vstupno-výstupného rozhrania na výstup HLS pre konfiguráciu BASIC 1

Počet vstupných vzoriek	**Rozhranie**	**Plocha [µm^2]**	**Latencia [počet cyklov]**	**Počet registrov**
16	sériové	34647	25,5	60
16	2 paralelne	45853	15	89
16	4 paralelne	48157	9	102
16	8 paralelne	60305	6	117
64	sériové	241007	97,5	332
64	2 paralelne	283662	51	380
64	4 paralelne	363065	27	404
64	8 paralelne	432990	15	401

Druhým hlavným faktorom pre odlišnú implementáciu algoritmu FFT je samotný *hls_config*, teda globálne využitie Datapath Optimizeru. Dopad na menšiu a väčšiu veľkosť algoritmu FFT bude opäť ukázaný pri 16 a 64 počte vstupných vzoriek pre vybrané vstupno-výstupné rozhrania v tab. 5.5.

Všeobecne je možné optimalizáciou dátových ciest dosiahnuť istú úsporu plochy na čipe, majú na to však vplyv ďalšie dva faktory. Týmito faktormi sú počet vstupných vzoriek a paralelizmus vstupných portov. Pre 8 vstupných vzoriek využitie DPA konfigurácie spôsobí zväčšenie plochy, pre 16 vzoriek sa pokles začne prejavovať až pri 8 paralelných vstupoch. Pre väčšie veľkosti FFT, ako sú 64 a 128 vstupných vzoriek, to je už pri 2 paralelných vstupoch. Pri sériovom vstupe plocha s využitím DPA konfigurácie v každom prípade stúpa. Menšia plocha bola dosiahnutá aj menším počtom registrov pri DPA návrhu, ktorý bol dosiahnutý skoro vo všetkých prípadom okrem niektorých výnimiek pri najmenšej veľkosti FFT – 8 vzoriek. Okrem niekoľko výnimiek (ako je napríklad práve 16 vzoriek pri 4 paralelných vstupoch) tiež ale platí, že konfigurácie s *dpopt_auto all* majú dlhšiu dátovú cestu.

Tabuľka 5.5 Vplyv konfigurácie *hls_config* na výstup HLS pri obmedzení latencie na 1 cyklus

Počet vstupných vzoriek	Rozhranie	*hls_config*	Plocha [µm^2]	Najdlhšia dátová cesta [ns]	Počet registrov
16	sériové	BASIC	34647	10,74	60
16	sériové	DPA	42095	10,85	52
16	2 paralelne	BASIC	45853	9,55	89
16	2 paralelne	DPA	45727	10,8	65
16	4 paralelne	BASIC	48157	12,91	102
16	4 paralelne	DPA	49591	10,4	87
16	8 paralelne	BASIC	60305	11,61	117
16	8 paralelne	DPA	50850	8,91	106
64	sériové	BASIC	241007	11,66	332
64	sériové	DPA	256235	13,55	281
64	2 paralelne	BASIC	283662	13,4	380
64	2 paralelne	DPA	248970	12,21	283
64	4 paralelne	BASIC	363065	12,18	404
64	4 paralelne	DPA	300764	13,37	323
64	8 paralelne	BASIC	432990	13,04	401
64	8 paralelne	DPA	314499	13,53	373

Ďalšia úspora plochy môže byť dosiahnutá zmiernením požiadaviek na výslednú latenciu. Z tab. 5.6 možno pozorovať, že táto možnosť sa dá efektívne využiť až pri vyšších veľkostiach vstupného vektora, konkrétne od 32 vzoriek. Paralelizmus v tomto prípade na veľkosť plochy nemal žiaden dopad, a pri ostatných parametroch výstupu nebola zistená žiadna výrazná závislosť.

Tabuľka 5.6 Vplyv obmedzenia cyklov na HLS pri konfigurácii BASIC so 4 vstupnými portami

Počet vstupných vzoriek	Limit hodinových cyklov	Plocha [µm^2]	Latencia [počet cyklov]
8	1	19701	6
8	2	19964	7
16	1	48157	9
16	2	50525	10
32	1	137299	15
32	2	131480	16
64	1	363065	27
64	2	339625	28
128	1	798188	51
128	2	729396	52

Posledný parameter, ktorý sa určitým spôsobom mení, je najdlhšia dátová cesta z hľadiska splnenia podmienky pre periódu hodinového signálu. Ku nesplneniu tejto požiadavky dochádza pri všetkých architektúrach s viac vstupnými vzorkami ako 16, iba pre 8 vzoriek je až na jednu výnimku táto časová hodnota dodržaná.

5.4 Verifikácia návrhu

Verifikácia, alebo aj overenie správnej funkčnosti návrhu, je v projekte programu Stratus integrované, ako už bolo naznačené v kapitole 4.1. Navrhnuté prostredie pre verifikáciu návrhu algoritmu FFT má teda tú istú štruktúru ako na obr. 4.1, SystemC modul tb sa tak skladá z dvoch procesov Clocked Thread obsahujúcich funkcie *source()* a *sink()*. Modul *tb* sa nachádza v prílohách B.4 a B.5.

Funkcia *source()* používa ako vstupný vektor vopred pripravené hodnoty uložené v súboroch v podpriečinku projektu s názvom *input_vectors*. V priečinku *golden* sa zas nachádzajú očakávané výstupné vektory pre daný počet vzoriek. Na základe počtu vzoriek je následne daný výstupný vektor skopírovaný do súboru *GOLDEN.dat*, ktorý je po získaní výstupných hodnôt z testovaného modulu s nimi porovnaný. Vzhľadom na použité rozhranie *cynw_p2p* sa podľa danej architektúry využívajú príslušné metódy na vsunutie vstupného vektora do testovaného modulu. Pre sériový vstup je to metóda *put()*, a pre paralelné rozhrania je to metóda *nb_put()* využitá v kombinácii s funkciou *cynw_wait_can_put()*, ktorá synchronizuje prístup ku všetkým vstupom.

Funkcia *sink()* má za úlohu získať výstupné hodnoty z testovaného modulu a zapisuje ich do výstupného súboru *output.dat.* Pre tento účel znovu využíva adekvátne metódy rozhrania *cynw_p2p* podľa danej architektúry – sériový vstup potrebuje metódu *get()* a paralelné vstupy kombináciu metódy *nb_get* s funkciou *cynw_wait_all_can_get()* zaisťujúcou synchronizáciu čítania výstupných hodnôt.

Porovnanie súborov *output.dat* a *GOLDEN.dat* prebieha za pomoci príkazu *compare* v *Makefile* súbore z prílohy B.10. V závere simulácie program používateľa informuje, či sa súbory zhodujú a výpočet teda prebehol v poriadku, alebo nie.

5.4.1 Výpočet latencie

Pre porovnanie jednotlivých architektúr bol do *tb* modulu vkomponovaný výpočet priemernej latencie. Získanie hodnoty latencie bolo prevedené za pomoci vzťahu:

$$lat_{avg} = \frac{lat_{total}}{samples} = \frac{\sum_{i=0}^{samples} \frac{(end_{im}[i] - start_{real}[i])}{clock}}{samples}, \tag{5.2}$$

kde lat_{total} je suma všetkých latencií jednotlivých výstupov a *samples* je počet vzoriek. Jednotlivé latencie sa pritom vypočítajú ako podiel rozdielu 2 časov a periódy hodinového signálu *clock*. Rozdiel v čitateli pritom tvorí získanie imaginárnej zložky výsledku $end_{im}[i]$ a vsunutie príslušného vstupu $start_{real}[i]$.

Časové údaje sú zo simulácie získané za pomoci funkcie zo SystemC knižnice *sc_time_stamp()* , ktorej návratová hodnota je aktuálny simulačný čas. Hodnota periódy hodinového signálu je vyčítaná z triedy *sc_clock*, ktorou je hodinový signál popísaný [7].

6 ZÁVER

Hlavnou náplňou tejto bakalárskej práce bolo odskúšať a zhodnotiť progres behaviorálnej syntézy praktickým návrhom algoritmu FFT. Pre tento účel bol vybraný nástroj z programového balíka od firmy Cadence – Stratus High-Level Synthesis. Pred samotnou syntézou bol najskôr navrhnutý behaviorálny referenčný model algoritmu FFT vo variante radix-2, aby bola overená správnosť výpočtu FFT. Už od začiatku návrhu sa pritom kládol dôraz na to, aby bol algoritmus jednoducho modifikovateľný pomocou zmien konštánt v návrhu.

Aby bol tento model vhodný pre HLS, muselo byť najskôr vykonaných niekoľko úprav. Nevyhnutnou modifikáciou bola manuálna konverzia klasických dátových typov *float* jazyka C na dátové typy z knižnice SystemC. Pri tomto kroku si treba dávať pozor, nakoľko môže veľmi ľahko vzniknúť drobná chyba v bitovom posuve, a návrh tak prestane korektne fungovať. Druhou časťou úprav bola náhrada používaných štandardných funkcií z knižnice *math.h*. Medzi tieto funkcie patrí výpočet druhej mocniny *pow2()* a logaritmu so základom dva *log2()*. Je zaujímavé, že oproti funkcii *log2()*, ktorá bola využitá na výpočet jednej konštanty, nemal Stratus problém identifikovať goniometrické funkcie *sin()* a *cos()* importované z tej istej knižnice. V tomto prípade Stratus dané výpočty vykonal dopredu a príslušné výsledky uložil do klasického poľa C++.

Po potrebných úpravách bol zistený dopad jednotlivých parametrov ovplyvňujúcich proces HLS. Rozdielne výsledky syntézy ukázali, že veľmi efektívnym parametrom pre úpravu časovania je *timing_aggression.* V súvislosti s týmto parametrom bolo potvrdené aj to, že Stratus môže niekedy navrhnúť časovanie prekračujúce hodnotu periódy hodinového signálu, pretože nástroj na logickú syntézu dokáže časovanie pozmeniť a čas najdlhšej dátovej cesty skrátiť. Ďalej je pre možnosť zdieľania prostriedkov potrebné povolenie rozvinutia slučiek *unroll_loops* a pre paralelný prístup k dátam sa hodí direktíva *flatten_arrays*. Určitá úspora plochy bola dosiahnutá aj s použitím optimalizácie výrazov *comm_subexp_elim*.

Naopak, žiaden dopad na návrh nemali optimalizačné direktívy *inline_partial_constants* či *balance_expr*. Výstup HLS tiež neovplyvnili agresívne plánovania *sched_aggresive_1* a *sched_aggresive_2*. V prípade navrhnutého algoritmu FFT tieto direktívy na dizajn nemali žiaden efekt, môžu však nájsť uplatnenie v iných dizajnoch.

Globálne nastavenia projektu sú však len jednou z možností, ako pozmeniť výslednú RTL podobu návrhu. Využité bolo aj lokálne nastavenie obmedzenia latencie výpočtu jadra FFT. Vypnutie tohto obmedzenia tiež ukázalo primárny cieľ optimalizácie behaviorálnej syntézy pomocou Stratusu – plochu návrhu. Tá má v tom prípade najvyššiu prioritu a latencia výpočtu je druhoradá. Pre optimalizáciu tohto parametru je preto potrebné využiť lokálne nastavenie *HLS_CONSTRAIN_LATENCY*.

Veľký vplyv na výslednú architektúru mala aj samotná podoba modulu, teda jeho vstupno-výstupné rozhranie. Práve pri rôznej miere paralelizmu Stratus ukázal svoju prispôsobivosť, kedy dokázal podľa počtu prichádzajúcich vstupných stimulov vytvoriť architektúry, ktoré obsahovali patričný počet kombinačnej logiky pre čo najnižšiu latenciu. Pre tieto konfigurácie *hls_config* BASIC bola tiež vytvorená alternatíva

v podobe konfigurácie DPA, ktorá využívala optimalizáciu dátových ciest pomocou integrovaného algoritmu programu Stratus – Datapath Optimizeru. Konfigurácie s týmto nastavením sa kladne prejavili najmä pri väčších počtoch vstupných vzoriek a vyššej miere paralelizmu. Pre algoritmy menšej veľkosti mal *hls_config* DPA väčšiu výslednú plochu, a okrem toho tiež spôsobuje zvýšenie najdlhšej dátovej cesty, teda maximálnu dosiahnuteľnú frekvenciu hodinového signálu.

Z pohľadu plochy návrhu a použitej technologickej knižnice môžu byť najmä architektúry FFT so 128 vstupnými vzorkami vyhodnotené ako nedostatočne optimalizované. RTL kód vytvorený skúseným návrhárom by určite mohol byť optimalizovaný lepšie, na druhej strane treba brať do úvahy čas, ktorý je tomu potrebný venovať.

Okrem týchto architektúr prebehli aj pokusy o vytvorenie viac špecifických hardvérových podôb algoritmu FFT. Architektúry s využitím spätnej väzby boli vytvorené za pomoci lokálnej *HLS_DPOPT_INLINE* direktívy, avšak tieto architektúry neboli dostatočne optimalizované. Problém nastal aj pri implementácii reťazového spracovávania, kedy všetky slučky vnútri algoritmu nemali podľa Stratusu konečný počet iterácií.

Z týchto skutočností vyplýva, že pre implementáciu špecifických architektúr je potrebné podľa toho prispôsobiť a viac pozmeniť vstupný kód. V prípade behaviorálnych algoritmov s absenciou časovania je lepšie ponechať nástroju Stratus väčšiu voľnosť pri voľbe architektúry. Ten je dostatočne flexibilný na vytvorenie RTL kódu s požadovanými parametrami s veľkou úsporou času. Pre nájdenie správnej konfigurácie syntézy sa určite vyplatí odskúšanie odlišných direktív, či už globálnych alebo lokálnych, záleží na povahe a požiadavkách dizajnu. Pokiaľ je to možné, k úprave architektúry dokáže veľmi prispieť aj zmena vstupno-výstupného rozhrania, teda počet stimulov, s ktorými môže dizajn pracovať.

Manuálny spôsob tvorby RTL podoby návrhu bude ešte nejakú dobu majoritnou metodikou návrhu digitálnych obvodov, avšak môžeme skonštatovať, že behaviorálna syntéza je veľmi zaujímavou alternatívou a správnym krokom vpred pre jednoduchšiu tvorbu komplexnejších dizajnov. Veľkou výhodou je tiež možnosť verifikácie viacerých úrovní navrhovaného modulu za pomoci prostredia vytvoreného s jazykom C++, ktoré tak ponúka široké možnosti a vysokú úroveň abstrakcie. Počas celého procesu návrhu ale treba mať neustále na pamäti, že samotná funkcia digitálneho obvodu síce môže byť opísaná behaviorálne, avšak výslednou implementáciou je stále fyzický hardvér.

LITERATÚRA

[1] MACK, Chris A. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing* [online]. 2011, **24**(2), 202 - 207 [cit. 2016-12-01]. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5696765&isnumber=5762426

[2] FIRGEROFF, Michael. *High-Level Synthesis Blue Book*. 1. United States of America: Xlibris, 2010. ISBN 1450097243.

[3] COUSSY, Philippe a Adam MORAWIEC. *High-level synthesis: from algorithm to digital circuit*. 1. Netherlands: Springer, 2008, xv, 297 s. ISBN 9781402085871.

[4] COUSSY, Philippe, Daniel D. GAJSKI, Michael MEREDITH a Andres TAKACH. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers* [online]. 2009, 26(4), 8-17 [cit. 2016-11-21]. DOI: 10.1109/MDT.2009.69. ISSN 0740-7475. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5209958&isnumber=5209950

[5] Functional Specification for SystemC 2.0: Update for SystemC 2.0.1. Version 2.0-Q. c1996-2002.

[6] Introduction to SystemC Part I. WELCOME TO WORLD OF ASIC [online]. India: Deepak Kumar Tala, 2014 [cit. 2016-11-25]. Dostupné z: http://www.asic-world.com/systemc/intro1.html#Introduction

[7] SystemC 2.0.1 Language Reference Manual. 1. San Jose: Open SystemC Initiative, c2003.

[8] SystemC Synthesizable Subset: Version 1.4.7. Elk Grove: Accellera Systems Initiative Inc., c2016.

[9] LYONS, Richard G. *Understanding digital signal processing*. Upper Saddle River: Prentice Hall, 2001. ISBN 9780201634679.

[10] SMÉKAL, Zdeněk. *Deterministické a náhodné signály pro integrovanou výuku VUT a VŠB-TUO* [online]. 1. Brno: Vysoké učení technické v Brně, 2013 [cit. 2016-12-03]. ISBN 978-80-214-4826-1. Dostupné z: https://vut-vsb.cz/predmet-analyza-signalu-a-soustav-46

[11] Signálové procesory v praxi: 6. přednáška - DFT, algoritmus FFT. *Katedra měření* [online]. Praha: České vysoké učení technické v Praze, c2008-2015 [cit. 2016-12-03]. Dostupné z: http://measure.feld.cvut.cz/system/files/files/cs/vyuka/predmety/A0M38SPP/slides/A0M38SPP_Prednaska_6.pdf

[12] MEYER-BAESE, Uwe. *Digital Signal Processing with Field Programmable Gate Arrays*. 3rd ed. Berlin: Springer, 2007, XX, 774. ISBN 978-3-540-72612-8.

[13] DVOŘÁK, Vojtěch. *Implementace výpočtu FFT v obvodech FPGA a ASIC*. Brno: Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií, 2013, 71 s.

[14] Archiv předmětu BASS Analýza signálů a soustav. *ITMULTI* [online]. Brno: Vysoké učení technické v Brně, c2016 [cit. 2016-12-03]. Dostupné z: https://archiv.itmulti.cz/46?archive-pwd=BASS%2FVideop%C5%99edn%C3%A1%C5%A1ky

[15] AYINALA, Manohar, Michael BROWN a Keshab K. PARHI. Pipelined Parallel FFT Architectures via Folding Transformation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* [online]. 2012, **20**(6), 1068-1081 [cit. 2016-12-04]. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5776727&isnumber=6196243

[16] Vladimir Stojanovic, course materials for 6.973 Communication System Design. *MIT OpenCourseWare*[online]. Cambridge (Massachusetts): Massachusetts Institute of Technology, 2006 [cit. 2016-12-04]. Dostupné z: https://ocw.mit.edu/courses/electrical-

engineering-and-computer-science/6-973-communication-system-design-spring-2006/lecture-notes/lecture_10.pdf

[17] *Stratus High-Level Synthesis* [online]. United States: Cadence, c2015 [cit. 2017-05-09]. Dostupné z: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/stratus-ds.pdf

[18] *Stratus HLS User Guide*. 16.14-s100. Cadence, 2016.

[19] *Stratus HLS Reference Guide*. 16.14-s100. Cadence, 2016.

[20] BRAUN, Axel G., Djones V. LETTNIN, Joachim GERLACH a Wolfgang ROSENSTIEL. Automated Conversion of SystemC Fixed-Point Data Types. *VLSI-SOC: From Systems to Chips* [online]. Boston: Kluwer Academic Publishers, 2006, (200), 55 [cit. 2017-05-14]. DOI: 10.1007/0-387-33403-3_4. ISBN 0-387-33402-5. Dostupné z: http://link.springer.com/10.1007/0-387-33403-3_4

ZOZNAM SYMBOLOV, VELIČÍN A SKRATIEK

ASIC	Aplication Specific Integrated Circuit, integrovaný obvod na zákazku
DFT	discrete Fourier transform, diskrétna Fourierova transformácia
DIF	Decimation-in-frequency, decimácia vo frekvenčnej oblasti
DIT	Decimation-in-time, Decimácia v čase
EDA	Electronic design automation, automatizácia elektronického návrhu
FFT	Fast Fourier Transform, rýchla Fourierova transformácia
FIFO	First in – first out, metóda spracovania dát v zásobníku
FPGA	Field-programmable gate array, programovateľné hradlové pole
GDS2	Graphic Database System, štandard prenosu výslednej podoby IO
HDL	hardware description language, jazyky slúžiace na popis hardvéru
HLS	High-level synthesis, behaviorálna syntéza
IO	integrovaný obvod
LSB	least significant bit, najmenej významný bit
MSB	most significant bit, najvýznamnejší bit
RTL	Register-transfer level, úroveň prenosov medzi registrami

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM PRÍLOH

A REFERENČNÝ MODEL ALGORITMU FFT

A.1 Model FFT v jazyku C++

```
#include <cmath>
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <ctime>
#include <iomanip>
#include "fft.h"
#define NUM_SAMPLES 64
using namespace std;

void fft(){

srand(time(0));
double samples[NUM_SAMPLES];
ofstream input_file;
input_file.open("input.txt");

//creating fft inputs with pseudo-random generator and writing to file
for(int i = 0; i< NUM_SAMPLES; i++){
      samples[i]=-5.0+double(rand())/double(RAND_MAX/(5+5));
      input_file<<std::fixed<<std::setprecision(51)<<samples [i]<<"\n";
      }
input_file.close();

//fill imaginary part of input with zeros, computing number of stages
double * samples_im;
samples_im = (double*) calloc(NUM_SAMPLES, sizeof(double));
int stages_cnt = int(log2(NUM_SAMPLES));
double samples_real[NUM_SAMPLES];

//bit reversal for cycle
for(int i = 0; i < NUM_SAMPLES; i++){
      int ind_reversed = 0;
      int ind_int = i;
      for(int j = (stages_cnt-1); j >= 0 ; j-- ){
      ind_reversed += ((ind_int%2) * pow(2, j));
      ind_int /= 2;
            }
      samples_real[ind_reversed] = samples[i];
      }
//stage cycle, computing number of section for curr_stage
for(int curr_stage=0; curr_stage < stages_cnt; curr_stage++){
      int section_cnt = pow(2, stages_cnt - (curr_stage + 1));

      //section cycle, computing number of butterflies for curr_sec
      for(int curr_sec=0; (curr_sec < section_cnt); curr_sec++){
            int bfly_cnt = int(pow(2, curr_stage));

            //butterfly cycle
            for(int curr_bfly=0;curr_bfly<bfly_cnt; curr_bfly++){
```

```
                    //computing indices for butterfly
                    int even=(pow(2,curr_stage+1)*curr_sec)+curr_bfly;
                    int odd = even + int(pow(2, curr_stage));

                    //computing twiddle factor
                    double twiddle_real = cos(((curr_bfly*section_cnt)* \
                          2*M_PI )/NUM_SAMPLES);
                    double twiddle_im = -sin(((curr_bfly* \
                    section_cnt)* 2 * M_PI )/NUM_SAMPLES);

                    //computing common results for butterfly outputs
                    double common_res_real = (twiddle_real * \
                    samples_real[odd]) - (samples_im[odd] * twiddle_im);
                    double common_res_im =(twiddle_real*samples_im[odd])\
                    + (twiddle_im * samples_real[odd]);

                    //computing butterfly outputs
                    samples_real[odd]=samples_real[even]-common_res_real;
                    samples_real[even] += common_res_real;
                    samples_im[odd] = samples_im[even] - common_res_im;
                    samples_im[even] +=  common_res_im;
               }
          }
}

//saving real and imaginary parts of outputs to results.txt file
ofstream res;
res.open("results.txt");

for(int i = 0; i < NUM_SAMPLES; i++){
      res<<std::fixed<<std::setprecision(51)<<samples_real[i]<<"\n";
      }
for(int i = 0; i < NUM_SAMPLES; i++){
      res<<std::fixed<<std::setprecision(51)<<samples_im[i]<<"\n";
      }

res.close();

}
```

A.2 Kontrola C++ modelu programom Matlab

```
%%opening files for internal fft computation and comparing results
fileID = fopen('input.txt', 'r');
formatSpec = '%f';
size = [1 Inf];
input = fscanf(fileID, formatSpec,size);
matlab_fft = fft(input);
real = real(matlab_fft);
im = imag(matlab_fft);
matlab_fft_vector = [real im];
fileID = fopen('results.txt', 'r');
c_fft = fscanf(fileID, formatSpec,size);
%%computing error of fft
compare = (matlab_fft_vector - c_fft);
```

B ZDROJOVÉ KÓDY V JAZYKU C++ - VARIANT S 2 VSTUPNÝMI A VÝSTUPNÝMI PORTAMI

B.1 Deklarácia FFT modulu – fft_two_samp.h

```
#ifndef __FFT_two_SAMP__H
#define __FFT_two_SAMP__H
//including necessary libraries
#include "systemc.h"
#include "cynw_p2p.h"
#include "math.h"
//defining constants used in design, which can be edited easily
#define NUM_SAMPLES 32
#define CONVERT 1024
#define PI 3.141592653
#define BIT_WIDTH 20
#define REAL_POS 0
#define IMAG_POS 1
#define COMMA_POS 10
#define IO_COUNT 8
//defining used data types in the design
typedef sc_int<20> used_type;
typedef sc_int<10> used_type_int;

SC_MODULE(fft_two_samp)
{
      //declaring input and outpurt ports of the design
      public:
      cynw_p2p < used_type  >::in     input1;
      cynw_p2p < used_type  >::in     input2;
      cynw_p2p < used_type  >::out    output1;
      cynw_p2p < used_type  >::out    output2;
      //declaring clock and reset signal
      sc_in_clk               clk;
      sc_in < bool >          rst;
      //module contructor
      SC_CTOR(fft_eight_samp):input1("input1"),input2("input2"),\
      output1("output1"),   output2("output2"),clk("clk"),   rst("rst"){
            //compute function is implemented as a Clocked Thread Process
            SC_CTHREAD(compute, clk.pos());
            reset_signal_is(rst,0);
            //connecting the clk and rst signals to the metaports
            input1.clk_rst(clk, rst);
            input2.clk_rst(clk, rst);
            output1.clk_rst(clk, rst);
            output2.clk_rst(clk, rst);
            }
      //functions inside fft_two_samp module
      void compute();
      int pow_2(int power);
      }
#endif
```

B.2 FFT modul – fft_two_samp.cpp

```
//including necessary libraries
#include "fft_two_samp.h"
#include "directives.h"

void fft_eight_samp::compute()
{
{ //reset the interfaces
HLS_DEFINE_PROTOCOL("reset");
input1.reset();
input2.reset();
output1.reset();
output2.reset();
wait();
}

//design behaviour
while (1)
{
used_type my_inputs[NUM_SAMPLES];
//input protocol - this part of code is different for every project!
for(used_type_int i = 0; i < (NUM_SAMPLES/IO_COUNT); i++){
      cynw_wait_all_can_get( input1, input2,);
      input1.nb_get(my_inputs[IO_COUNT*i]);
      input2.nb_get(my_inputs[IO_COUNT*i + 1]);
      }
//end of different section of code
used_type_int ind_reversed = 0;
used_type_int ind_int = 0;
used_type constants_real[NUM_SAMPLES/2];
used_type constants_im[NUM_SAMPLES/2];
used_type samples_int = NUM_SAMPLES;
used_type_int stages_cnt = 0;
used_type my_outputs[NUM_SAMPLES][2];

//twiddle factors computation
for(used_type_int i =0; i < (NUM_SAMPLES/2); i++){
      constants_real[i] = used_type(cos((i*2*PI)/NUM_SAMPLES)*CONVERT);
      constants_im[i] = used_type(-sin((i*2*PI)/NUM_SAMPLES)*CONVERT);
      }
{
//macros for local directives for synthesis saved in directives.h
MAIN_LATENCY_B;
MAIN_LATENCY_D;
//computation of the number of stages
while(samples_int != 1){
      samples_int /= 2;
      stages_cnt++;
      }
//saving inputs into output array in bit reversed order
for(used_type_int i = 0; i < NUM_SAMPLES; i++){
      ind_int = i;
      for(used_type_int j = (stages_cnt-1); j >= 0 ; j-- ){
      ind_reversed += ((ind_int%2) * pow_2(j));
      ind_int /= 2;
      }
      my_outputs[ind_reversed][REAL_POS] = my_inputs[i];
      my_outputs[i][IMAG_POS] = 0;
      ind_reversed = 0;
      }
```

```
//main FFT loop
for(used_type_int curr_stage = 0;curr_stage < stages_cnt; curr_stage++){
      //loop for each stage of FFT butterfly diagram
      used_type_int section_cnt = pow_2(stages_cnt - (curr_stage + 1));
      for(used_type_int curr_sec = 0;curr_sec < section_cnt;curr_sec++){
            //loop for each section in current stage
            used_type bfly_cnt = pow_2(curr_stage);
            for(used_type_int curr_bfly=0; curr_bfly < bfly_cnt;\
                  curr_bfly++){
                  //loop for each simple 2 input butterfly
                  used_type_int even = (pow_2(curr_stage + 1) *\
                  curr_sec) + curr_bfly;
                  used_type_int odd = even + bfly_cnt;
                  used_type_int con = curr_bfly * section_cnt;
                  //computing common results of the output
                  used_type real_mult_1 = (sc_int<BIT_WIDTH>)\
                  ((( sc_int<BIT_WIDTH*2> )( constants_real[con] )) *\
                  (( sc_int<BIT_WIDTH*2> )(my_outputs[odd][REAL_POS]))\
                  >> COMMA_POS );

                  used_type real_mult_2 = (sc_int<BIT_WIDTH>)\
                  ( ((sc_int<BIT_WIDTH*2>)(my_outputs[odd][IMAG_POS]))\
                  * ((sc_int<BIT_WIDTH*2>)(constants_im[con])) >>\
                  COMMA_POS);
                  used_type common_res_real = real_mult_1 - real_mult_2;

                  used_type im_mult_1 = ((sc_int<BIT_WIDTH>)\
                  ( ((sc_int<BIT_WIDTH*2>) (constants_real[con])) *\
                  ((sc_int<BIT_WIDTH*2>)(my_outputs[odd][IMAG_POS]))>>\
                  COMMA_POS));
                  used_type im_mult_2 = ((sc_int<BIT_WIDTH>)\
                  ((sc_int<BIT_WIDTH*2 >)(constants_im[con])) *\
                  ((sc_int<BIT_WIDTH*2>)(my_outputs[odd][REAL_POS]))>>\
                  COMMA_POS);
                  used_type common_res_im = im_mult_1 + im_mult_2;
                  //computing butterfly outputs
                  my_outputs[odd][REAL_POS] =\
                  my_outputs[even][REAL_POS] - common_res_real;
                  my_outputs[even][REAL_POS] =\
                  my_outputs[even][REAL_POS] + common_res_real;
                  my_outputs[odd][IMAG_POS] =\
                  my_outputs[even][IMAG_POS] - common_res_im;
                  my_outputs[even][IMAG_POS] =\
                  my_outputs[even][IMAG_POS] + common_res_im;
                  }
            }
      }
}

// Output protocol - this part of code is different for every project!
for(used_type_int i = 0; i < NUM_SAMPLES/(IO_COUNT/2); i++){
      cynw_wait_can_put( output1, output2,);
      output1.nb_put(my_outputs[(IO_COUNT/2)*i][REAL_POS]);
      output2.nb_put(my_outputs[(IO_COUNT/2)*i][IMAG_POS]);
      }
//end of different section of code
//end of an infinite while() loop
}
//end of a compute() function
}
```

```
// Function for substitution of pow2() in the design
int fft_eight_samp::pow_2(int power)
{
int result = 1;
if (power == 0){
      return result;
      }
else{
      for(used_type_int i = 0; i < power; i++){
            result *= 2;
            }
      return result;
      }
}
```

B.3 Lokálne direktívy – directives.h

```
#ifndef DIRECTIVES_H
#define DIRECTIVES_H
//this directive is applied in case hls_config BASIC is used
#ifdef BASIC
#define MAIN_LATENCY_B HLS_CONSTRAIN_LATENCY( 2, 2, "my_latency" )
#else
#define MAIN_LATENCY_B
#endif
//this directive is applied in case hls_config DPA is used
#ifdef DPA
#define MAIN_LATENCY_D HLS_CONSTRAIN_LATENCY( 2, 2, "my_latency" )
#else
#define MAIN_LATENCY_D
#endif

#endif
```

B.4 Deklarácia modulu tb – tb.h

```
#ifndef __TB__H
#define __TB__H
//including necessary libraries
#include "cynw_p2p.h"
#include "fft_two_samp.h"
#include "systemc.h"

SC_MODULE(tb)
{
      //declaring input and outpurt ports of the module
      public:
      cynw_p2p < used_type >::base_in    input1;
      cynw_p2p < used_type >::base_in    input2;
      cynw_p2p < used_type >::base_out   output1;
      cynw_p2p < used_type >::base_out   output2;
      //declaration of clock and reset parameters
      sc_in_clk                    clk;
      sc_out < bool >              rst;
      sc_in < bool >               rst_in; // sampling version of "rst"
      //module contructor with processes and definition of reset signal
      SC_CTOR(tb){
            SC_CTHREAD(source, clk.pos());
```

```
            SC_CTHREAD(sink, clk.pos());
            reset_signal_is(rst_in,0);
            rst_in(rst);
            }
      //declaring functions inside the module
      void source();
      void sink();
      //file pointers used in the module
      FILE *input_file;
      FILE *outfp;
      FILE *golden_current;
      FILE *golden_source;
      //variables used for computing average latency
      sc_time start_time[NUM_SAMPLES],end_time[NUM_SAMPLES*2]
      sc_time clock_period;
};
#endif
```

B.5 Verifikačný modul – tb.cpp

```
#include "tb.h"
//source thread
void tb::source()
{
//reset the output metaport and cycle the design's reset
output1.reset();
output2.reset();
rst = 0;
wait(2);
rst = 1;
wait();
//arrays for storing input vectors and char ch for copying files
int values_int[NUM_SAMPLES];
used_type values[NUM_SAMPLES];
char ch;
//according to number of samples correct golden file is chosen and copied
//to GOLDEN.dat file
if(NUM_SAMPLES == 8){
      golden_current = fopen( "./golden/GOLDEN_8.dat", "r" );
      golden_source = fopen( "./golden/GOLDEN.dat", "w" );
      while(1){
            ch = fgetc(golden_current);
            if(ch == EOF)
                  break;
            else
                  putc(ch, golden_source);
            }
      fclose(golden_current);
      fclose(golden_source);
      //correct input file is opened
      input_file = fopen( "./input_vectors/input_vector_8.dat", "r" );
      }

if(NUM_SAMPLES == 16){
      golden_current = fopen( "./golden/GOLDEN_16.dat", "r" );
      golden_source = fopen( "./golden/GOLDEN.dat", "w" );
      while(1){
            ch = fgetc(golden_current);
            if(ch == EOF)
```

```
                        break;
                else
                        putc(ch, golden_source);
                }
        fclose(golden_current);
        fclose(golden_source);
        //correct input file is opened
        input_file = fopen( "./input_vectors/input_vector_16.dat", "r" );
        }

if(NUM_SAMPLES == 32){
        golden_current = fopen( "./golden/GOLDEN_32.dat", "r" );
        golden_source = fopen( "./golden/GOLDEN.dat", "w" );
        while(1){
                ch = fgetc(golden_current);
                if(ch == EOF)
                        break;
                else
                        putc(ch, golden_source);
                }
        fclose(golden_current);
        fclose(golden_source);
        //correct input file is opened
        input_file = fopen( "./input_vectors/input_vector_32.dat", "r" );
        }

if(NUM_SAMPLES == 64){
        golden_current = fopen( "./golden/GOLDEN_64.dat", "r" );
        golden_source = fopen( "./golden/GOLDEN.dat", "w" );
        while(1){
                ch = fgetc(golden_current);
                if(ch == EOF)
                        break;
                else
                        putc(ch, golden_source);
                }
        fclose(golden_current);
        fclose(golden_source);
        //correct input file is opened
        input_file = fopen( "./input_vectors/input_vector_64.dat", "r" );
        }

if(NUM_SAMPLES == 128){
        golden_current = fopen( "./golden/GOLDEN_128.dat", "r" );
        golden_source = fopen( "./golden/GOLDEN.dat", "w" );
        while(1){
                ch = fgetc(golden_current);
                if(ch == EOF)
                        break;
                else
                        putc(ch, golden_source);
                }
        fclose(golden_current);
        fclose(golden_source);
        //correct input file is opened
        input_file = fopen( "./input_vectors/input_vector_128.dat", "r" );
        }
//condition for case file was not opened correctly
if(input_file == NULL){
        printf( "Error Reading File\n" );
        }
```

```
//getting values from the input file and converting them to integer
for(int i = 0; i < NUM_SAMPLES; i++){
      fscanf(input_file, "%d\n", &values_int[i]);
      values[i] = (used_type(values_int[i]*CONVERT));
      }
//closing the input file
fclose(input_file);
//sending input values into design module
//this part of code is different for each project!
//simulation times are also saved for computing latency
for(int i = 0; i < NUM_SAMPLES/IO_COUNT; i++){
      cynw_wait_can_put( output1, output2 );
      output1.nb_put(values[IO_COUNT*i]);
      start_time[IO_COUNT*i] = sc_time_stamp();
      output2.nb_put(values[IO_COUNT*i+1]);
      start_time[IO_COUNT*i+1] = sc_time_stamp();
      }//end of a different part of code
//this code prevents simulation from hanging with sc_stop() function
wait(10000);
printf( "Hanging simulation stopped. Please check DUT module. \n" );
sc_stop();
}

//read all the expected values from the design
void tb::sink()
{
//getting value of clock period from sc_clock class
sc_clock *clk_p = DCAST<sc_clock*>(clk.get_interface());
clock_period = clk_p->period();
//opening output.dat file and checking if the procedure was successful
outfp = fopen( "./output.dat ", "w" );
if(outfp == NULL){
      printf( "Couldn't open file for writing" );
      }

double total_cycles = 0;
input1.reset();
input2.reset();
wait();
//to synchronize with reset
used_type results[NUM_SAMPLES*2];
//getting the values from the design modules
//this part of code is different for each project!
//simulation times are also saved for computing latency
 for(int i = 0; i < (NUM_SAMPLES); i++){
      cynw_wait_all_can_get( input1, input2 );
      input1.nb_get(results[IO_COUNT*i]);
      fprintf( outfp, "%d\n", (int)results[IO_COUNT*i]);
      end_time[IO_COUNT*i] = sc_time_stamp();
      input2.nb_get(results[(IO_COUNT*i)+1]);
      fprintf( outfp, "%d\n", (int)results[(IO_COUNT*i)+1]);
      end_time[IO_COUNT*i+1] = sc_time_stamp();
      }//end of a different part of code
//loop for printing results and computing total cycles
for(int i = 0; i < (NUM_SAMPLES * 2); i++){
      double res_double = (results[i].to_double()) / CONVERT;
      printf( "%f \n", res_double);
      if(i < NUM_SAMPLES){
            total_cycles += ((end_time[2*i+1] - start_time[i]) / \
            clock_period);
```

```
            }
      }
//getting the average latency as a division of total cycles and clock
//period (according to the formula in chapter 5.4.1
printf(" \nAverage latency is %g cycles.\n " \, (double)(total_cycles\
/ NUM_SAMPLES));
//ending the simulation run
fclose(outfp);
sc_stop();
}
```

B.6 Deklarácia hlavného modulu TOP – system.h

```
#ifndef SYSTEM_H_INCLUDED
#define SYSTEM_H_INCLUDED
//including necessary libraries
#include <systemc.h>
#include <esc.h>
#include "cynw_p2p.h"

#include "tb.h"
#include "fft_two_samp.h"
#include "fft_two_samp_wrap.h"

SC_MODULE(TOP)
{
      public:
      //cynw_p2p channels for connection of tb and design module
      cynw_p2p < used_type >      input1_chan;
      cynw_p2p < used_type >      input2_chan;
      cynw_p2p < used_type >     output1_chan;
      cynw_p2p < used_type >     output2_chan;
      //clock and reset signals
      sc_clock            clk;
      sc_signal < bool >  rst;
      //the testbench and DUT modules instances
      fft_two_samp_wrapper *m_dut;
      tb   *m_tb;
      //functions for creating and deleting instances
      void initInstances();
      void deleteInstances();
      //module constructor
      SC_CTOR(TOP):clk("clk", CLOCK_PERIOD, SC_NS, 0.5, 0, SC_NS, true),
            input1_chan( "input1_chan" ),
            input2_chan( "input2_chan" ),
            output1_chan( "output1_chan" ),
            output2_chan( "output2_chan" ),
            rst( "rst" )
      {
            initInstances();
      }

      ~TOP()
      {
            deleteInstances();
      }
};
#endif
```

B.7 Hlavný modul system – system.cpp

```
#include "system.h"

void TOP::initInstances()
{
	//connecting the design module
	m_dut = new fft_two_samp_wrapper( "fft_two_samp_wrapper" );

	m_dut->clk(clk);
	m_dut->rst(rst);
	m_dut->input1(input1_chan);
	m_dut->output1(output1_chan);
	m_dut->input2(input2_chan);
	m_dut->output2(output2_chan);

	//connecting the testbench
	m_tb = new tb("tb");

	m_tb->clk(clk);
	m_tb->rst(rst);
	m_tb->output1(input1_chan);
	m_tb->input1(output1_chan);
	m_tb->output2(input2_chan);
	m_tb->input2(output2_chan);
}

void TOP::deleteInstances()
{
	delete m_tb;
	delete m_dut;
}
```

B.8 Funkcia na spustenie simulácie – main.cpp

```
#include "system.h"

TOP *top = NULL;
//function for creating new instance of the top module
extern void esc_elaborate()
{
	top = new TOP("top");
}
//function for deleting instance of the top module
extern void  esc_cleanup()
{
	delete top;
}

int sc_main(int argc, char *argv[])
{
	esc_initialize(argc, argv);
	//starting simulation
	esc_elaborate();
	sc_start();
	return 0;
}
```

B.9 Projektový súbor project.tcl

```
# Project Parameters
#
# Technology Libraries
#
set LIB_PATH "/home/users/stud49/libraries/FreePDK45/osu_soc/lib/files"
set LIB_LEAF "gscl45nm.lib"
use_tech_lib "$LIB_PATH/$LIB_LEAF"

#uncomment for the waveform debugging
#use_systemc_simulator incisive
#setting global attributes for Stratus
set_attr cc_options " -DCLOCK_PERIOD=10.0"
set_attr hls_cc_options " -DCLOCK_PERIOD=10.0"
set_attr clock_period 10
set_attr sched_aggressive_1 on
set_attr unroll_loops on
set_attr flatten_arrays all
set_attr timing_aggression 0
set_attr end_of_sim_command "make compare"
set_attr comm_subexp_elim yes
#system modules (not synthesizable)
define_system_module main main.cpp
define_system_module system system.cpp
define_system_module tb tb.cpp
#module for high-level synthesis
define_hls_module fft_two_samp fft_two_samp.cpp
#defining hls_configs for hls_module fft_two_samp
define_hls_config fft_two_samp BASIC
define_hls_config fft_two_samp DPA
#defining sim_configs for hls_module fft_two_samp in several setup
define_sim_config B {fft_two_samp}
define_sim_config V_BASIC {fft_two_samp RTL_V BASIC}
define_sim_config V_DPA {fft_two_samp RTL_V DPA}
#alternative way of setting global parameter, this time it is for
#hls_configs that start with DP*
set_attr dpopt_auto all [find -hls_config DP*]
```

B.10 Súbor Makefile

```
saySimPassed:
	@bdw_sim_pass

BDW_DEBUG = 1
#including generated Makefile.prj in the top-level Makefile
-include Makefile.prj
#generating Makefile.prj from project.tcl file
Makefile.prj : project.tcl
	@bdw_makegen

#clean out undesirable files from the project directory
clean: clean_all clean_libs
	@rm -f transcript vsim* *.wlf data.out
	@rm -f *~ *.bak *.BAK
	@rm -rf work debussy* vfast*
	@rm -f Makefile.prj
	@rm -f msg_help.html
	@rm -rf core
```

```
	@rm -rf core.*
	@rm -rf .stack.*

CLEAN: clean
#comparing GOLDEN.dat reference file with output.dat file
compare:
	@echo "****************************************"
	@if cmp -s ./golden/GOLDEN.dat ./output.dat ; then \
	echo "SIMULATION PASSED"; \
	else \
	echo "SIMULATION FAILED"; \
	fi
	@echo "****************************************"
```

C SÚHRNNÉ VÝSLEDKY SYNTÉZY MODULOV FFT

C.1 Počet vzoriek N = 8

Rozhranie	*hls config*	Plocha [µm^2]	Latencia [hodinové cykly]	Max. dátová cesta [ns]	Počet násobičiek	Počet sčítačiek	Počet odčítačiek	Počet registrov
sériové	BASIC 1	12815	13,5	6,21	1	2	1	30
sériové	BASIC 2	12412	14,5	4,94	1	2	1	22
sériové	DPA 1	17016	13,5	5,5	-	-	-	28
sériové	DPA 2	17332	14,5	5,5	-	-	-	29
2 paralelne	BASIC 1	17121	9	4,69	2	2	2	42
2 paralelne	BASIC 2	14288	10	5,93	1	2	2	41
2 paralelne	DPA 1	18923	9	5,52	-	-	-	35
2 paralelne	DPA 2	19231	10	5,48	-	-	-	36
4 paralelne	BASIC 1	19701	6	7,11	2	4	4	53
4 paralelne	BASIC 2	19964	7	6,23	2	3	2	57
4 paralelne	DPA 1	20985	6	5,52	-	-	-	51
4 paralelne	DPA 2	20870	7	5,48	-	-	-	52
8 paralelne	BASIC 1	21113	4,5	10,2	2	5	4.1	76
8 paralelne	BASIC 2	21288	5,5	9,33	2	4	3	79
8 paralelne	DPA 1	20793	4,5	6,71	-	-	-	74
8 paralelne	DPA 2	20583	5,5	6,83	-	-	-	78

C.2 Počet vzoriek N = 16

Rozhranie	*hls config*	Plocha [µm^2]	Latencia [hodinové cykly]	Max. dátová cesta [ns]	Počet násobičiek	Počet sčítačiek	Počet odčítačiek	Počet registrov
sériové	BASIC 1	34647	25,5	10,74	3	4	2	60
sériové	BASIC 2	30973	26,5	6,52	2	5	2	60
sériové	DPA 1	42095	25,5	10,85	-	-	-	52
sériové	DPA 2	42377	26,5	8,47	-	-	-	52
2 paralelne	BASIC 1	45853	15	9,55	5	9	5	89
2 paralelne	BASIC 2	46427	16	12,6	6	6	3	82
2 paralelne	DPA 1	45727	15	10,8	-	-	-	65
2 paralelne	DPA 2	45557	16	10,75	-	-	-	64
4 paralelne	BASIC 1	48157	9	12,91	6	9	6	102
4 paralelne	BASIC 2	50525	10	8,44	7	9	5	101

4 paralelne	DPA 1	49591	9	10,4	-	-	-	87
4 paralelne	DPA 2	48562	10	8,39	-	-	-	84
8 paralelne	BASIC 1	60305	6	11,61	9	13	10	117
8 paralelne	BASIC 2	58327	7	11,82	8	11	10	129
8 paralelne	DPA 1	50850	6	8,91	-	-	-	106
8 paralelne	DPA 2	50511	7	8,24	-	-	-	115

C.3 Počet vzoriek N = 32

Rozhranie	***hls config***	**Plocha [µm^2]**	**Latencia [hodinové cykly]**	**Max. dátová cesta [ns]**	**Počet násobičiek**	**Počet sčítačiek**	**Počet odčítačiek**	**Počet registrov**
sériové	BASIC 1	100838	49,5	12,14	9	15	6	154
sériové	BASIC 2	88119	50,5	12,95	6	14	6	145
sériové	DPA 1	107185	49,5	13,08	-	-	-	119
sériové	DPA 2	98732	50,5	9,38	-	-	-	113
2 paralelne	BASIC 1	110098	27	12,95	11	17	14	178
2 paralelne	BASIC 2	111016	28	12,83	11	18	14	185
2 paralelne	DPA 1	110672	27	13,43	-	-	-	137
2 paralelne	DPA 2	104422	28	11,96	-	-	-	147
4 paralelne	BASIC 1	137299	15	12,97	19	33	20	202
4 paralelne	BASIC 2	131480	16	13,06	17	30	16	205
4 paralelne	DPA 1	116229	15	10,84	-	-	-	154
4 paralelne	DPA 2	118349	16	12,67	-	-	-	163
8 paralelne	BASIC 1	157452	9	12,33	24	45	30	217
8 paralelne	BASIC 2	138879	10	12,7	19	33	23	239
8 paralelne	DPA 1	124737	9	11,93	-	-	-	186
8 paralelne	DPA 2	121204	10	13,45	-	-	-	194

C.4 Počet vzoriek N = 64

Rozhranie	***hls config***	**Plocha [µm^2]**	**Latencia [hodinové cykly]**	**Max. dátová cesta [ns]**	**Počet násobičiek**	**Počet sčítačiek**	**Počet odčítačiek**	**Počet registrov**
sériové	BASIC 1	241007	97,5	11,66	20	41	21	332
sériové	BASIC 2	225242	98,5	12,2	16	36	19	315
sériové	DPA 1	256235	97,5	13,55	-	-	-	281
sériové	DPA 2	235146	98,5	13,41	-	-	-	277

2 paralelne	BASIC 1	283662	51	13,4	31	63	29	380
2 paralelne	BASIC 2	259526	52	12,8	23	47	36	371
2 paralelne	DPA 1	248970	51	12,21	-	-	-	283
2 paralelne	DPA 2	247477	52	13,2	-	-	-	295
4 paralelne	BASIC 1	363065	27	12,18	52	82	55	404
4 paralelne	BASIC 2	339625	28	12,53	45	65	49	406
4 paralelne	DPA 1	300764	27	13,37	-	-	-	323
4 paralelne	DPA 2	294784	28	13,42	-	-	-	339
8 paralelne	BASIC 1	432990	15	13,04	75	119	62	401
8 paralelne	BASIC 2	370711	16	13,4	58	88	59	446
8 paralelne	DPA 1	314499	15	13,53	-	-	-	373
8 paralelne	DPA 2	310874	16	13,15	-	-	-	374

C.5 Počet vzoriek N = 128

Rozhranie	***hls config***	**Plocha [µm^2]**	**Latencia [hodinové cykly]**	**Max. dátová cesta [ns]**	**Počet násobičiek**	**Počet sčítačiek**	**Počet odčítačiek**	**Počet registrov**
sériové	BASIC 1	512790	193,5	11,92	35	66	35	667
sériové	BASIC 2	495529	194,5	11,37	30	58	33	658
sériové	DPA 1	545075	193,5	13,15	-	-	-	597
sériové	DPA 2	538599	194,5	12,95	-	-	-	620
2 paralelne	BASIC 1	621250	99	12,81	60	89	58	738
2 paralelne	BASIC 2	600288	100	11,62	54	88	54	731
2 paralelne	DPA 1	598062	99	13,1	-	-	-	640
2 paralelne	DPA 2	569237	100	13,31	-	-	-	646
4 paralelne	BASIC 1	798188	51	13,43	106	164	136	787
4 paralelne	BASIC 2	729396	52	13,19	91	122	104	823
4 paralelne	DPA 1	692221	51	13,4	-	-	-	696
4 paralelne	DPA 2	674783	52	13,39	-	-	-	706
8 paralelne	BASIC 1	1099785	27	13,32	184	317	208	757
8 paralelne	BASIC 2	966482	28	13,14	138	255	177	877
8 paralelne	DPA 1	814654	27	13,61	-	-	-	689
8 paralelne	DPA 2	732615	28	13,64	-	-	-	722